Data Mining Project Phase 1

Stroke Prediction

Team Members :

Ali Ashraf Hamdy – 23P0233
Omar Fouad Abdelhamid – 23P0146
Seif El-Din Tamer Safwat – 23P0240
Asaad Ramzy Asaad – 23P0228
Adel Waheed Mohamed – 23P0213
Kirollous Ramzy – 22P0194
Ahmed Tarek – 22P0301

AIN SHAMS UNIVERISTY

Introduction

Stroke is one of the leading causes of death and long-term disability worldwide. Early detection of individuals at high risk of experiencing a stroke can significantly improve prevention and treatment outcomes. With the rise of healthcare data and machine learning, predictive models have become powerful tools for identifying key risk factors and forecasting potential health issues.

This project aims to **analyze patient health records**, perform **data preprocessing**, and build **machine learning models** capable of predicting the likelihood of a stroke. The project also explores data visualization, outlier detection, feature engineering, and classification algorithms to understand both the structure of the data and the patterns behind stroke occurrences.

Dataset Description

The dataset used in this project is the **Stroke Prediction Dataset**, which contains various health-related attributes for a group of individuals. Each record represents a single person, with demographic information, lifestyle details, and medical measurements. The dataset includes the following features:

Demographic Features

- **gender** – Male or Female.
- **age** – Age of the individual.
- **ever_married** – Whether the individual has ever been married.
- **Residence_type** – Rural or Urban residency.

Lifestyle and Social Factors

- **work_type** – Type of occupation (Private, Self-employed, Govt job, Children, Never worked).
- **smoking_status** – Current smoking habits (formerly smoked, never smoked, smokes, unknown).

Medical Attributes

- **hypertension** – Indicates if the person has hypertension (1 = yes, 0 = no).

- **heart_disease** – Indicates if the person has a heart condition (1 = yes, 0 = no).
- **avg_glucose_level** – Average blood glucose level.
- **bmi** – Body Mass Index.

<> Target Variable

- **stroke** – Indicates whether the individual had a stroke (1 = yes, 0 = no).
  This is a **binary classification** target.

Dataset types

```
id                                int64
gender                            int32
age                             float64
hypertension                      int64
heart_disease                     int64
ever_married                      int32
Residence_type                    int32
avg_glucose_level               float64
bmi                             float64
stroke                            int64
avg_glucose_level_log           float64
work_type_Never_worked             bool
work_type_Private                  bool
work_type_Self-employed            bool
work_type_children                 bool
smoking_status_formerly smoked     bool
smoking_status_never smoked        bool

smoking_status_smokes              bool
```

Data Understanding Phase

This phase focuses on **loading the dataset**, **examining its structure**, and gaining an initial idea of its contents. The code contributes to this phase in the following way:

1. Loading the Dataset

The `pd.read_csv()` function brings the dataset into the environment so it can be inspected and analyzed.
This is the first step in understanding what data you are working with.

2. Inspecting Dataset Size

The command `df.shape` provides the dimensions of the dataset:

- Number of rows (records or observations)
- Number of columns (features or attributes)

Knowing the dataset size is essential to:

- Understand the scale of the problem
- Prepare for cleaning, preprocessing, and modeling
- Identify if the dataset is large, small, or requires adjustments

```python
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
```

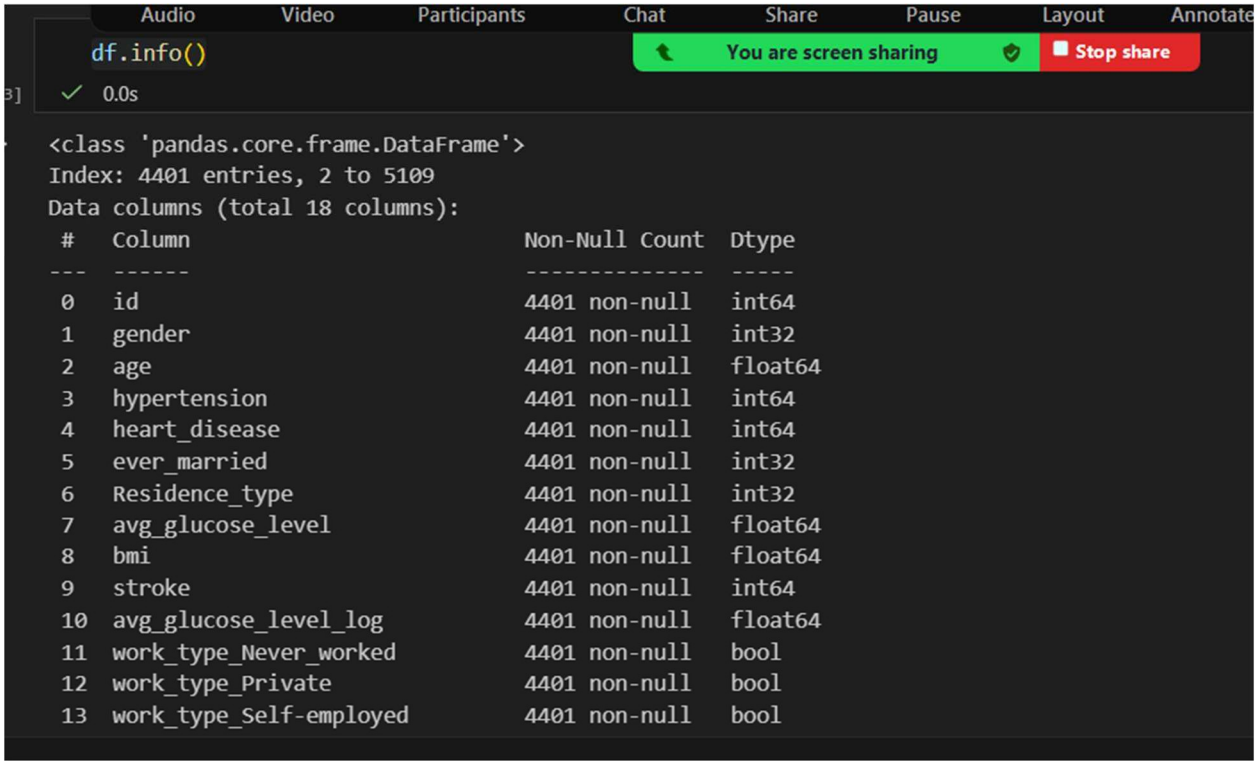
```python
df=pd.read_csv("C:\\Term 5\\Data Mining\\project\\healthcare-dataset-stroke-data.csv")
```

```python
df.head()
```

```python
df.shape
```

```
our dataset contain 12  columns and 5110 rows

# features types
## numeric
age
```

```
avg_glucose_level

bmi
## boolean and categorical

gender

hypertension

heart_disease

ever_married

work_type

Residence_typr

smoking_status

stroke
```



df.info()

This command displays a summary of the dataset, including:

- Column names
- Number of non-null (non-missing) values
- Data types of each column
- Total number of entries

Purpose:
It helps identify which columns contain missing values and what data types (e.g., integers, floats, objects) need to be handled during preprocessing.

```
df.describe() # no indicateion for outliers but need more analysis as mean approxamienly equal median
```

[34] ✓ 0.1s

	id	gender	age	hypertension	heart_disease	ever_married	Residence_type	avg_glucose_level	bmi
count	4401.000000	4401.000000	4401.000000	4401.000000	4401.000000	4401.00000	4401.000000	4401.000000	4401.000000
mean	36616.904567	0.409680	41.285190	0.074074	0.039082	0.62304	0.507612	91.483933	27.798887
std	21151.541938	0.492292	22.171772	0.261921	0.193812	0.48468	0.499999	22.663934	6.614072
min	67.000000	0.000000	1.000000	0.000000	0.000000	0.00000	0.000000	55.120000	10.300000
25%	17893.000000	0.000000	23.000000	0.000000	0.000000	0.00000	0.000000	75.070000	23.200000
50%	37011.000000	0.000000	42.000000	0.000000	0.000000	1.00000	1.000000	88.040000	27.400000
75%	54866.000000	1.000000	58.000000	0.000000	0.000000	1.00000	1.000000	104.030000	31.900000
max	72940.000000	2.000000	82.000000	1.000000	1.000000	1.00000	1.000000	168.680000	47.500000

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS JUPYTER Filter (e.g. te

`df.describe()`

This function generates descriptive statistics for **numerical** columns, such as:

- Count
- Mean
- Standard deviation
- Minimum and maximum values
- Quartiles (25%, 50%, 75%)

Purpose:
It provides an initial understanding of the distribution of numerical data, detects unusual values, and highlights potential outliers.

```
for column in df.select_dtypes(include=['object']).columns:
    print(f"{column}: {df[column].nunique()} unique values")
```

3. Counting Unique Values in Categorical Columns

Explanation:

- `df.select_dtypes(include=['object'])` selects all categorical columns.
- `df[column].nunique()` counts how many distinct values each categorical column contains.
- The loop prints the number of unique categories for each categorical feature.

Purpose:
This step helps determine:

- How many categories each feature has
- Whether a column has too many or too few categories
- What type of encoding (Label Encoding or One-Hot Encoding) is appropria

`df.duplicated()`

- This function checks each row in the dataset and determines whether it is a **duplicate** of a previous row.
- It returns a series of **True/False** values:
  - **True** → the row is duplicated
  - **False** → the row is unique

`.sum()`

- Since **True = 1** and **False = 0**, summing the results gives the **total number of duplicated rows** in the dataset.

Target Variable Analysis

Understanding the distribution of the target variable (**stroke**) is an essential step before building machine learning models. Since this is a **classification problem**, analyzing the target helps identify whether the dataset is balanced or imbalanced, which affects the choice of evaluation metrics and preprocessing techniques such as SMOTE or class weighting.

The following code visualizes the distribution of the target variable:

```
plt.figure(figsize=(5,4))
sns.countplot(x="stroke", data=df)
plt.title("Distribution of Stroke Cases")
plt.xlabel("Stroke (0 = No, 1 = Yes)")
plt.ylabel("Count")
plt.show()
#most cases doesnt have stroke
```

1. `plt.figure(figsize=(5,4))`

Creates a new figure for the plot with a defined width and height.
This ensures the visualization is displayed clearly and proportionally.

2. `sns.countplot(x="stroke", data=df)`

This line creates a **count plot** using Seaborn.
A count plot shows how many times each category appears.

- `x="stroke"` → The target variable on the x-axis
- `data=df` → The dataset to use

This allows us to see how many individuals had a stroke (1) versus those who did not (0).

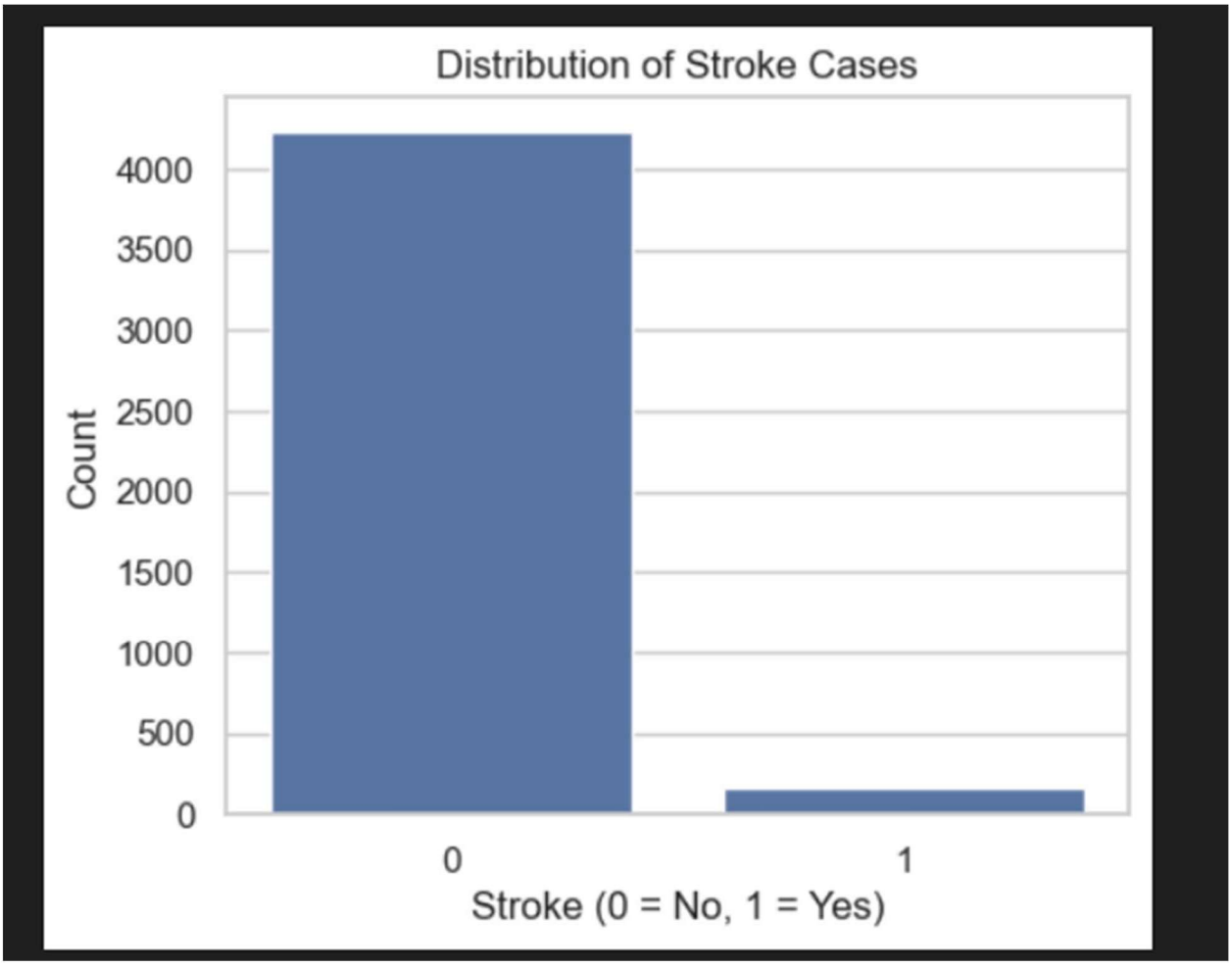
3. Labels and Title

- `plt.title()` → Adds a title to the chart
- `plt.xlabel()` → Labels the x-axis
- `plt.ylabel()` → Labels the y-axis

These labels make the plot easy to interpret.

4. `plt.show()`

Displays the final plot.



`df['stroke']`

This selects the **stroke** column from the dataset.
This column contains the target variable with values:

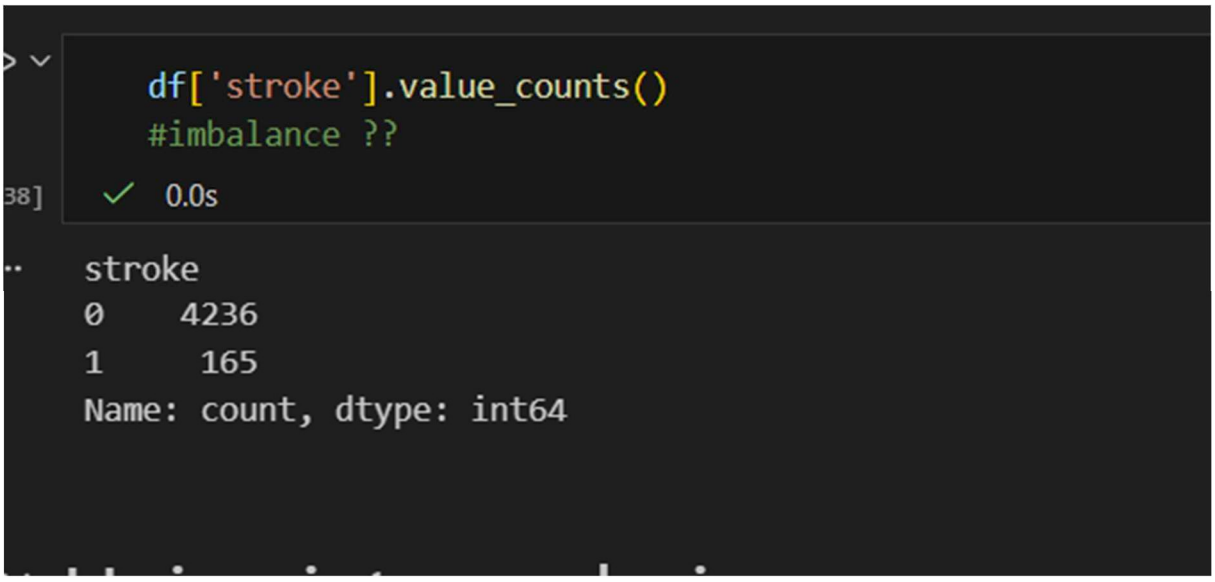
- **0** → No stroke
- **1** → Stroke

`value_counts()`

This function counts how many times each unique value appears in the selected column.

It returns:

- The number of people who **did not** have a stroke (0)
- The number of people who **did** have a stroke (1)



Univariate Analysis

Univariate analysis focuses on examining each variable individually to understand its distribution, structure, and statistical characteristics. This step is essential in the **Data Understanding** phase of the data mining process, as it helps identify patterns, detect outliers, and understand the nature of both numerical and categorical features before applying further preprocessing or modeling.

Below is the full analysis for both **numerical** and **categorical** features.

---

1. Univariate Analysis for Numerical Features

```
num_features = ['age', 'bmi', 'avg_glucose_level']

# Set visual style
sns.set(style="whitegrid")

for col in num_features:
    plt.figure(figsize=(6,4))
    sns.histplot(df[col], kde=True, bins=30, color='skyblue',
edgecolor='black')
    plt.title(f'Distribution of {col}', fontsize=13)
    plt.xlabel(col.capitalize())
    plt.ylabel('Count')
    plt.show()
```

Code Explanation

1. Selecting numerical features

```
num_features = ['age', 'bmi', 'avg_glucose_level']
```

These are the key numerical variables in the dataset. Each will be analyzed separately.

2. Setting visualization style

```
sns.set(style="whitegrid")
```

This improves the appearance of the plots for readability.

3. Looping through each numerical feature

The loop creates a separate **histogram** for each numerical column:

- `sns.histplot()` → Plots the distribution of the variable
- `kde=True` → Adds a smooth probability density curve
- `bins=30` → Controls the granularity of the histogram
- `edgecolor='black'` → Makes bars clearer

📌 Purpose of This Analysis

- Understand the distribution (normal, skewed, uniform)
- Identify outliers or extreme values
- Check for potential transformations (log scaling)
- Detect imbalanced ranges in variables such as glucose or BMI

🔹 2. Univariate Analysis for Categorical Features (Tabular Summary)

```
cat_features = [
    'gender',
    'hypertension',
    'heart_disease',
    'ever_married',
    'work_type',
    'Residence_type',
    'smoking_status',
    'stroke'  # include target for quick overview
]

for col in cat_features:
    print(f"\n===== {col.upper()} =====")
    counts = df[col].value_counts()
    percentages = df[col].value_counts(normalize=True) * 100
    summary = pd.DataFrame({'Count': counts, 'Percentage':
percentages.round(2)})
    print(summary)
```

📌 Code Explanation

1. Selecting categorical features

A list of all categorical variables, including the target variable.

2. `value_counts()`

Counts how many times each category appears.

3. `value_counts(normalize=True)`

Computes percentages instead of counts.

4. Creating a summary table

A DataFrame is created showing:

- **Count** of each category
- **Percentage** representation

📌 Purpose of This Analysis

- Understand category frequency distribution
- Identify rare categories
- Detect imbalanced categorical variables
- Inform decisions about encoding (One-Hot, Label Encoding)

🔷 3. Visualizing Categorical Features (Count Plots)

A. First Group of Categorical Features

```
cat_features_1 = ['gender', 'hypertension', 'heart_disease']

for col in cat_features_1:
    plt.figure(figsize=(6,4))
    sns.countplot(x=col, data=df, palette='Set2', edgecolor='black',
hue=None, legend=False)
    plt.title(f'Distribution of {col}', fontsize=13)
    plt.xlabel(col.capitalize())
    plt.ylabel('Count')
    plt.xticks(rotation=30)
    plt.show()
```

Explanation

Each feature in this list is plotted using a **count plot**, which visualizes how many times each category appears. This helps detect:

- Imbalances (e.g., most people have no hypertension)
- Distribution patterns
- Whether categories are evenly spread

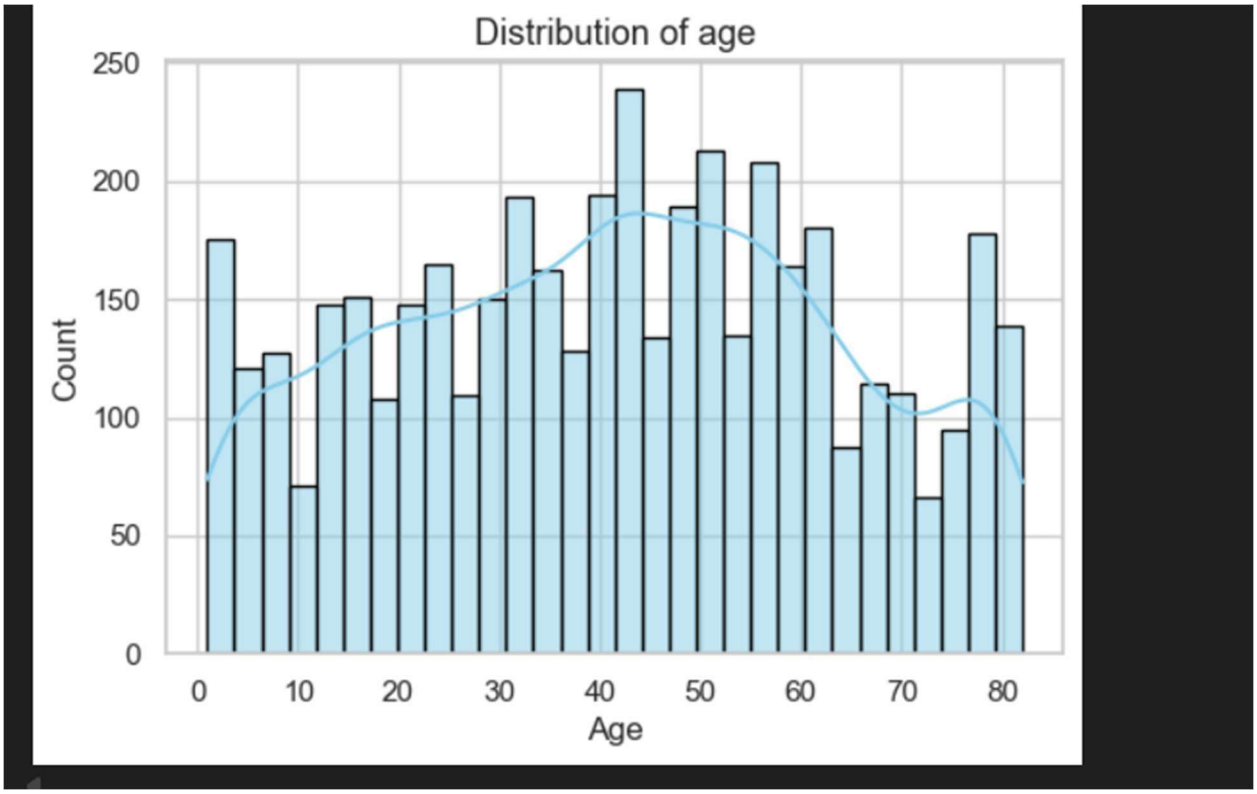
B. Second Group of Categorical Features

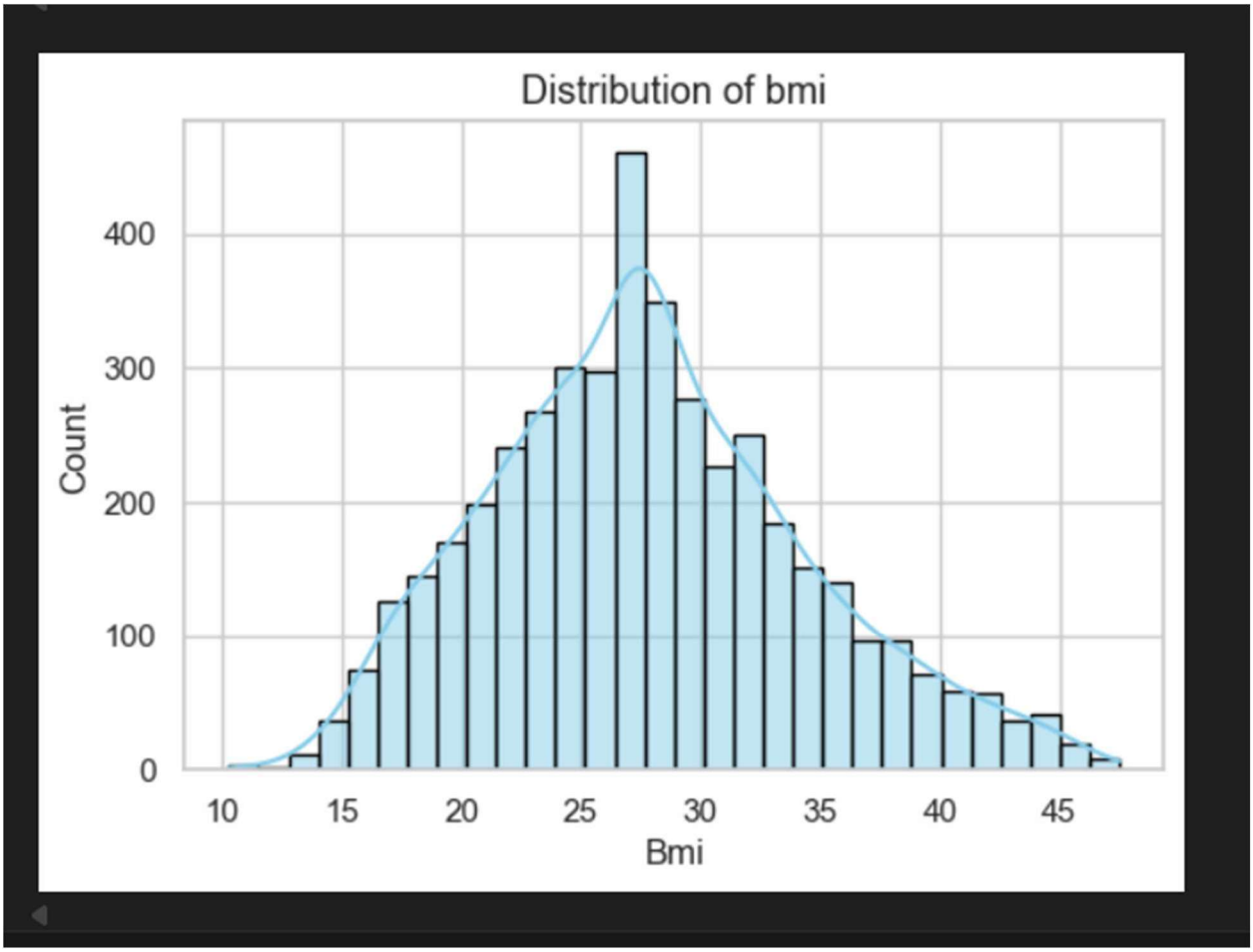
```
cat_features_2 = ['ever_married', 'work_type']

for col in cat_features_2:
    plt.figure(figsize=(6,4))
    sns.countplot(x=col, data=df, palette='Set2', edgecolor='black',
hue=None, legend=False)
    plt.title(f'Distribution of {col}', fontsize=13)
    plt.xlabel(col.capitalize())
    plt.ylabel('Count')
    plt.xticks(rotation=30)
    plt.show()
```

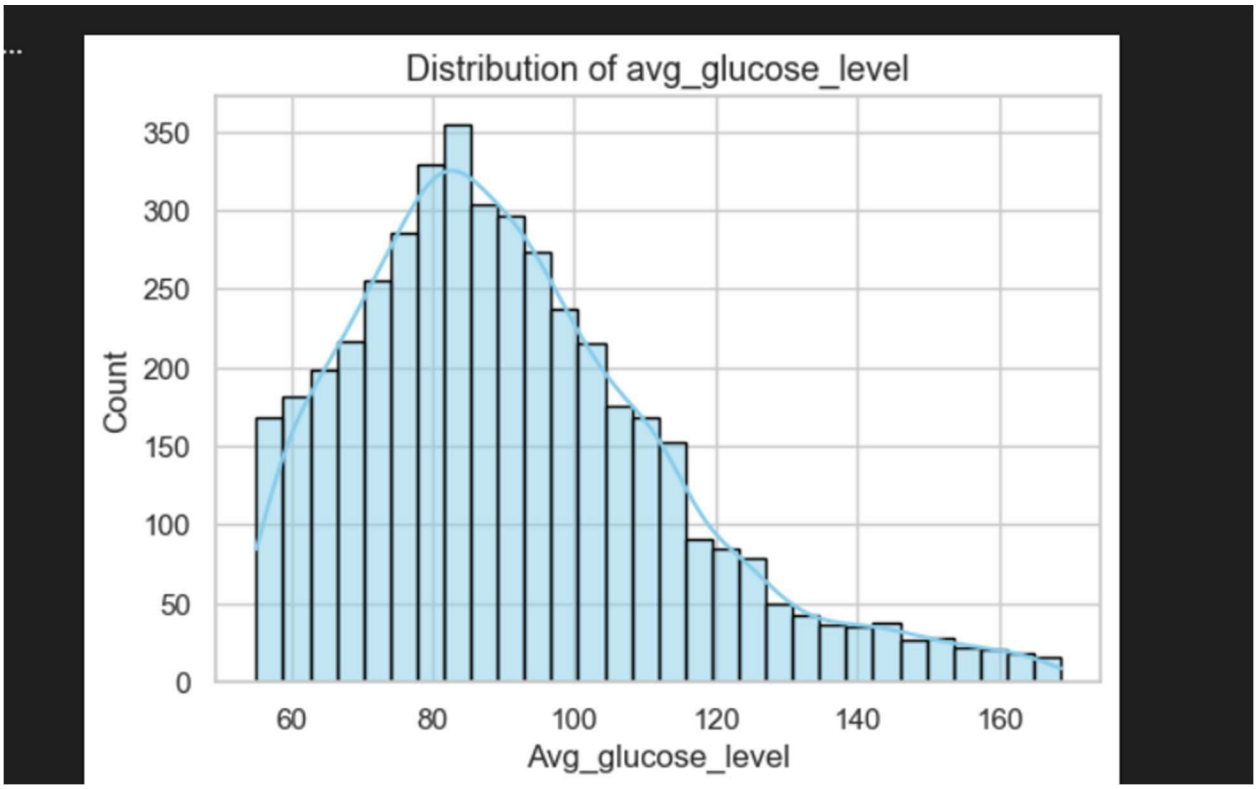
Explanation

Same approach as the previous block, but for different features.
Rotating the x-axis labels improves readability, especially for `work_type`.

```
num_features = ['age', 'bmi', 'avg_glucose_level']

# Set visual style
sns.set(style="whitegrid")

for col in num_features:
    plt.figure(figsize=(6,4))
    sns.histplot(df[col], kde=True, bins=30, color='skyblue', edgecolor='black')
    plt.title(f'Distribution of {col}', fontsize=13)
    plt.xlabel(col.capitalize())
    plt.ylabel('Count')
    plt.show()
```


Distribution of age
250
200
150
100
50
0
Count
0
10
20
30
40
50
60
70
80
Age

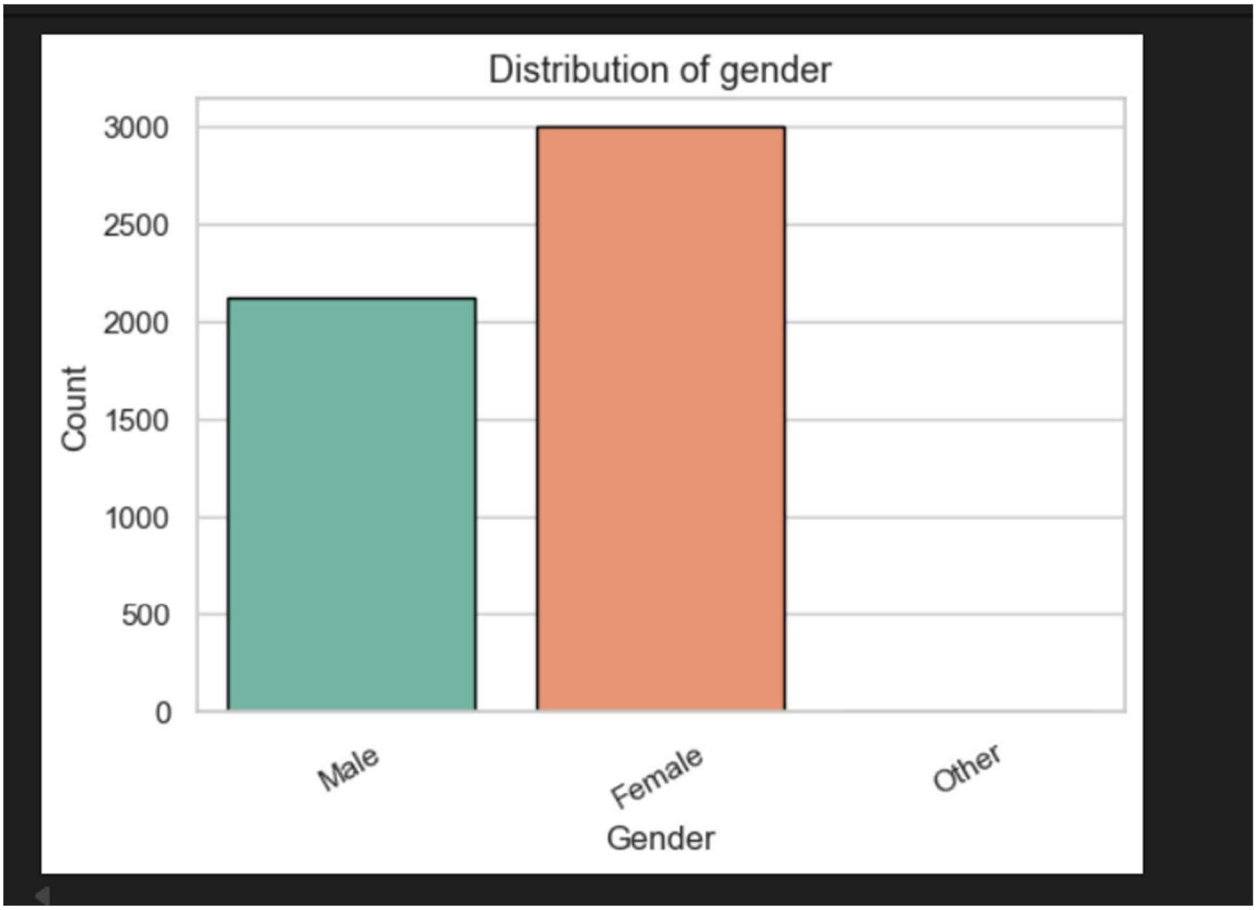

Distribution of bmi
Count
400
300
200
100
0
10
15
20
25
30
35
40
45
Bmi

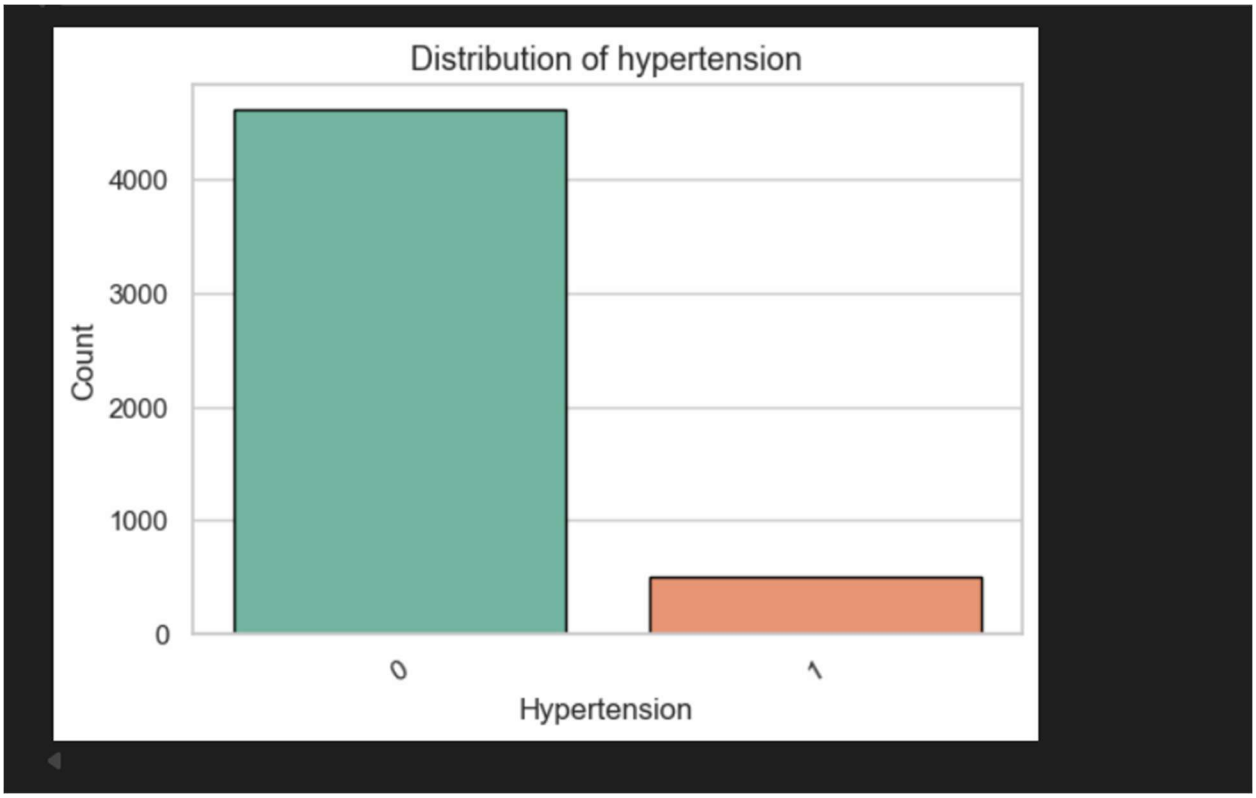


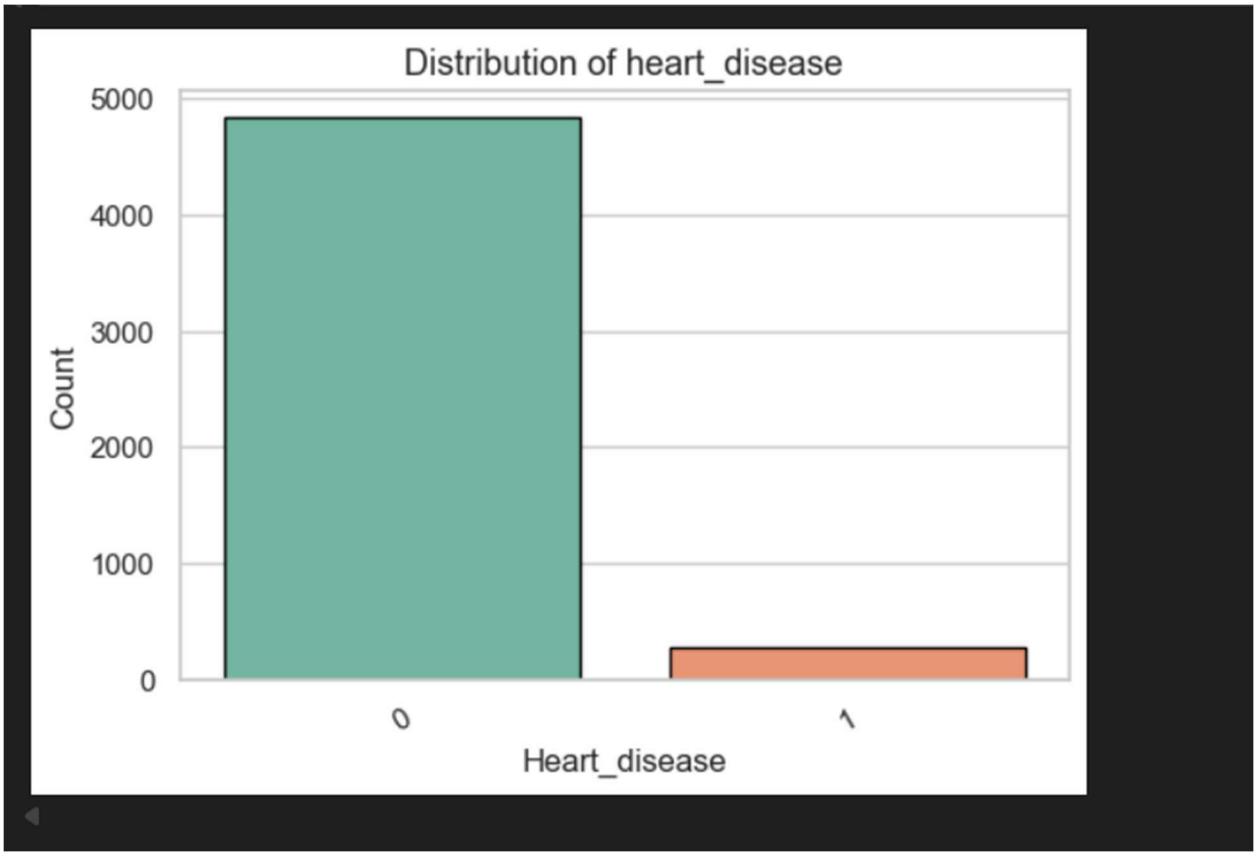

```
for col in cat_features:
    print(f"\n===== {col.upper()} =====")
    counts = df[col].value_counts()
    percentages = df[col].value_counts(normalize=True) * 100
    summary = pd.DataFrame({'Count': counts, 'Percentage': percentages.round(2)})
    print(summary)
```

```
===== GENDER =====
        Count  Percentage
gender
Female   2994       58.59
Male     2115       41.39
Other       1        0.02

===== HYPERTENSION =====
              Count  Percentage
hypertension
0              4612       90.25
1               498        9.75

===== HEART_DISEASE =====
               Count  Percentage
heart_disease
0               4834        94.6
1                276         5.4

===== EVER_MARRIED =====
              Count  Percentage
ever_married
Yes            3353       65.62
No             1757       34.38
```

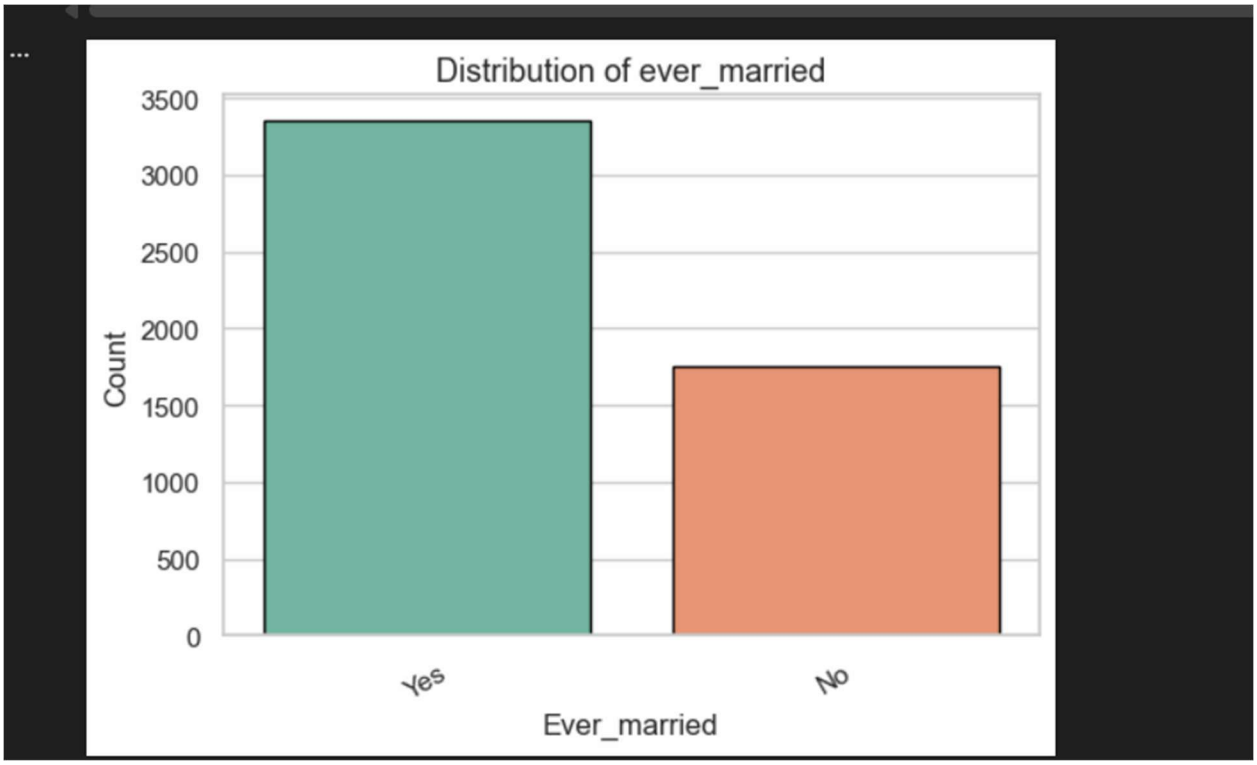
```
cat_features_1 = ['gender', 'hypertension', 'heart_disease']

for col in cat_features_1:
    plt.figure(figsize=(6,4))
    sns.countplot(x=col, data=df, palette='Set2', edgecolor='black', hue=None,
legend=False)
    plt.title(f'Distribution of {col}', fontsize=13)
    plt.xlabel(col.capitalize())
    plt.ylabel('Count')
    plt.xticks(rotation=30)
    plt.show()
```


Distribution of gender
3000
2500
2000
1500
1000
500
0
Count
Male
Female
Other
Gender


Distribution of hypertension
4000
3000
2000
1000
0
Count
0
1
Hypertension




```
cat_features_2 = ['ever_married', 'work_type']

for col in cat_features_2:
    plt.figure(figsize=(6,4))
    sns.countplot(x=col, data=df, palette='Set2', edgecolor='black', hue=None,
legend=False)
    plt.title(f'Distribution of {col}', fontsize=13)
    plt.xlabel(col.capitalize())
    plt.ylabel('Count')
    plt.xticks(rotation=30)
    plt.show()
```



1. Categorical Features Distribution

```
cat_features_3 = ['Residence_type', 'smoking_status', 'stroke']

for col in cat_features_3:
    plt.figure(figsize=(6,4))
    sns.countplot(x=col, data=df, palette='Set2', edgecolor='black',
hue=None, legend=False)
    plt.title(f'Distribution of {col}', fontsize=13)
    plt.xlabel(col.capitalize())
    plt.ylabel('Count')
    plt.xticks(rotation=30)
    plt.show()
```

Explanation

- **Purpose:** Visualize the distribution of key categorical variables including `Residence_type`, `smoking_status`, and the target `stroke`.
- **`sns.countplot()`**: Plots the frequency of each category.
- **Observations:**
  - Helps identify category imbalances.
  - Shows which categories dominate among stroke and non-stroke cases.

◇ 2. Numerical Features vs Target Variable

```
num_features = ['age', 'bmi', 'avg_glucose_level']

for col in num_features:
    plt.figure(figsize=(6,4))
```

```
    sns.boxplot(x='stroke', y=col, data=df, color='skyblue')
    plt.title(f'{col} vs Stroke')
    plt.xlabel('Stroke')
    plt.ylabel(col.capitalize())
    plt.show()
```

Explanation

- **Purpose:** Examine how numerical features vary across the target classes (`stroke = 0` or `1`).
- **`sns.boxplot()`**:
  - Visualizes distribution, median, quartiles, and outliers.
  - `x='stroke'` separates the data by target class.
  - `y=col` plots the numerical variable values.
- **Observations:**
  - Identify if people who experienced a stroke differ significantly in age, BMI, or glucose level compared to those who did not.
  - Outliers can be detected for potential treatment.

<> 3. Mean Comparison Table

```
print(df.groupby('stroke')[num_features].mean())
```

Explanation

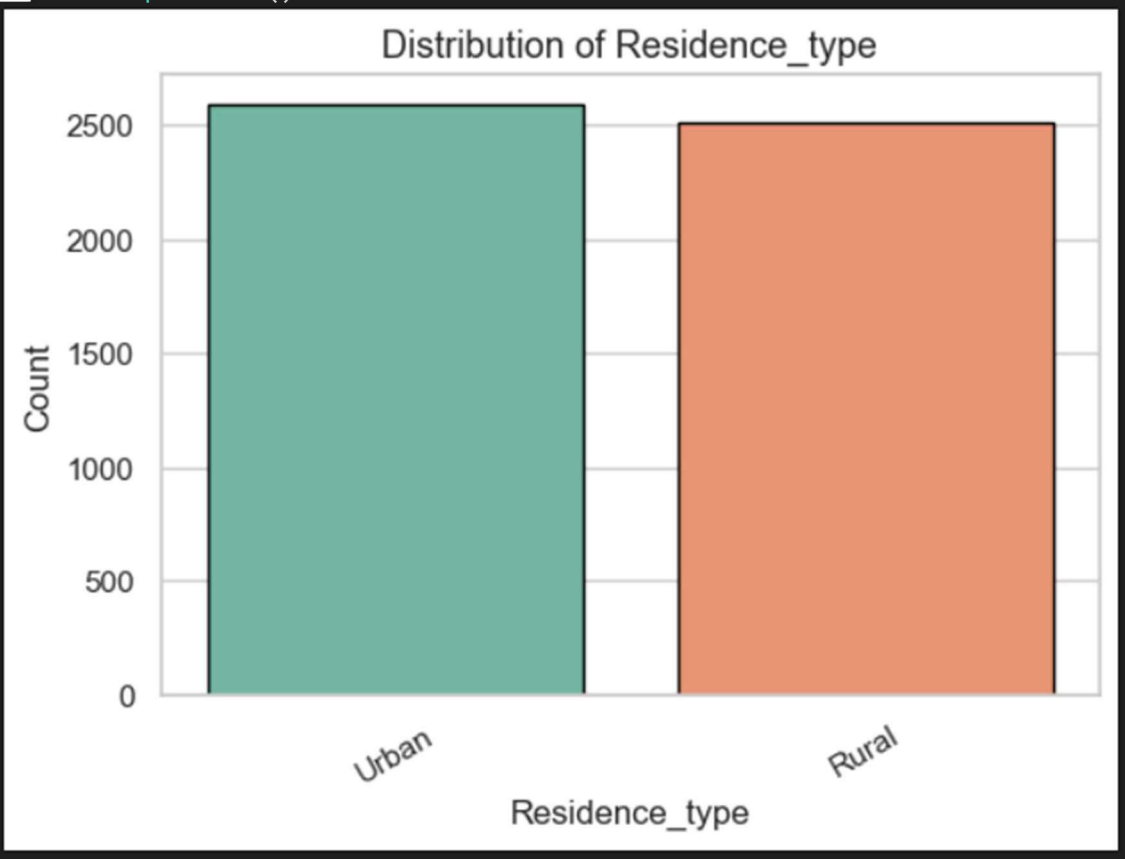
- **`df.groupby('stroke')`**: Groups the dataset by the target variable (`stroke`).
- **`[num_features].mean()`**: Calculates the mean of each numerical feature for each group.
- **Purpose:**
  - Provides a quick statistical comparison between stroke and non-stroke individuals.
  - Helps identify features that may be strong predictors of stroke.

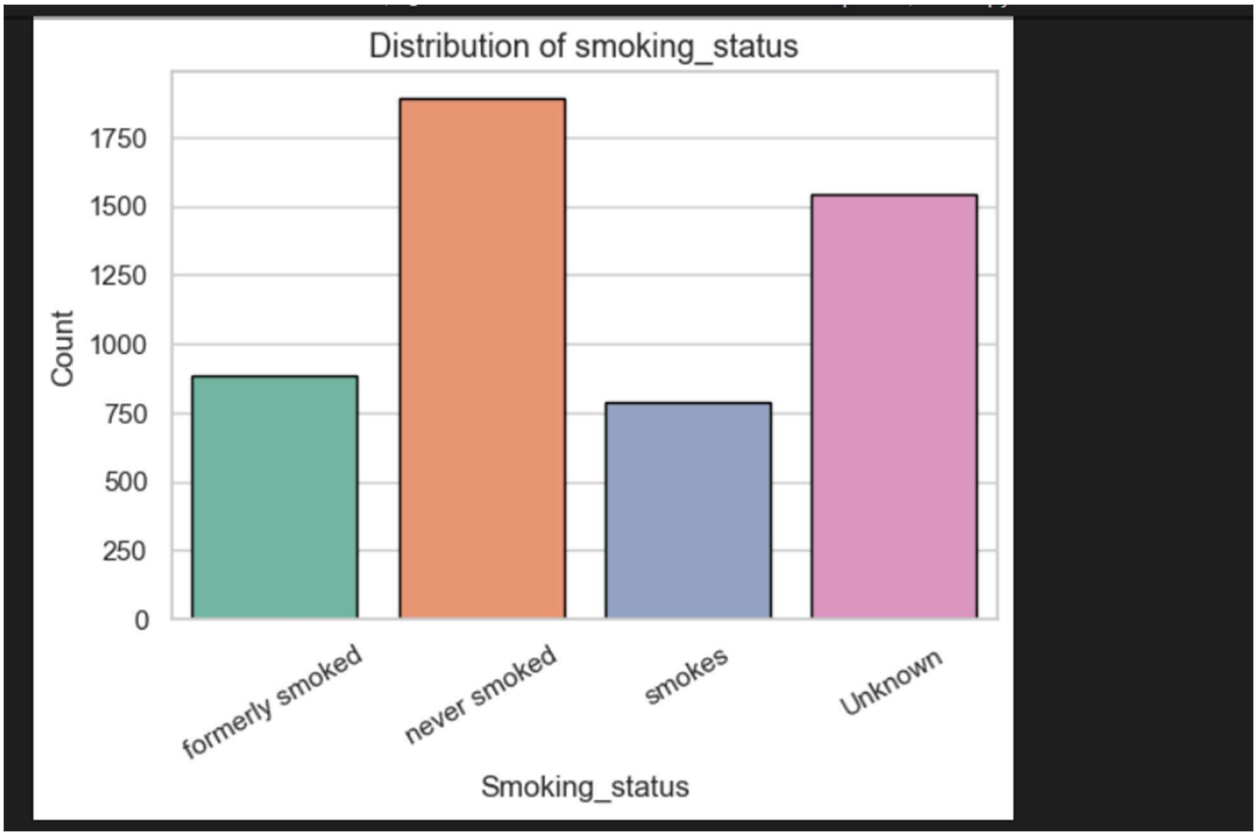
Example Interpretation:

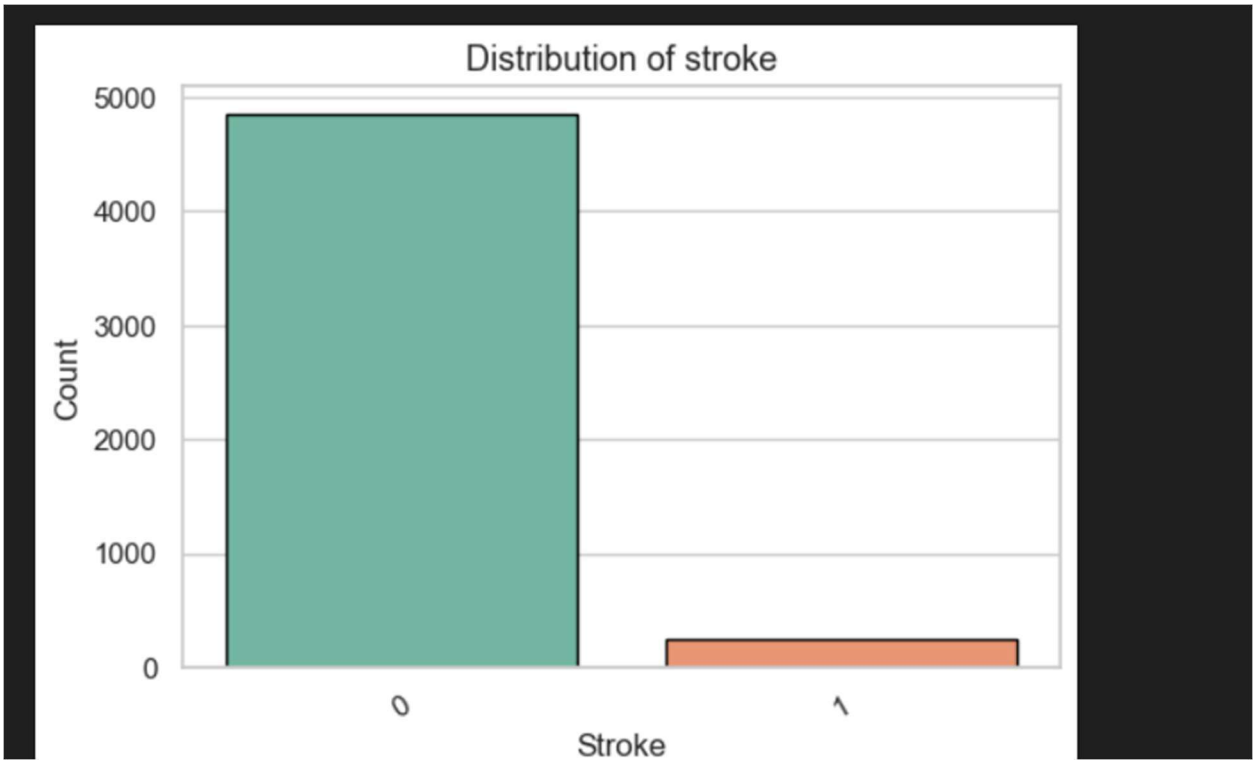
- If the mean age of stroke patients is higher than non-stroke patients, age may be a strong risk factor.
- Similarly, higher average glucose levels or BMI in stroke cases may indicate relevant health patterns.

```
cat_features_3 = ['Residence_type', 'smoking_status', 'stroke']

for col in cat_features_3:
    plt.figure(figsize=(6,4))
    sns.countplot(x=col, data=df, palette='Set2', edgecolor='black', hue=None, legend=False)
    plt.title(f'Distribution of {col}', fontsize=13)
    plt.xlabel(col.capitalize())
    plt.ylabel('Count')
    plt.xticks(rotation=30)
```

```
plt.show()
```

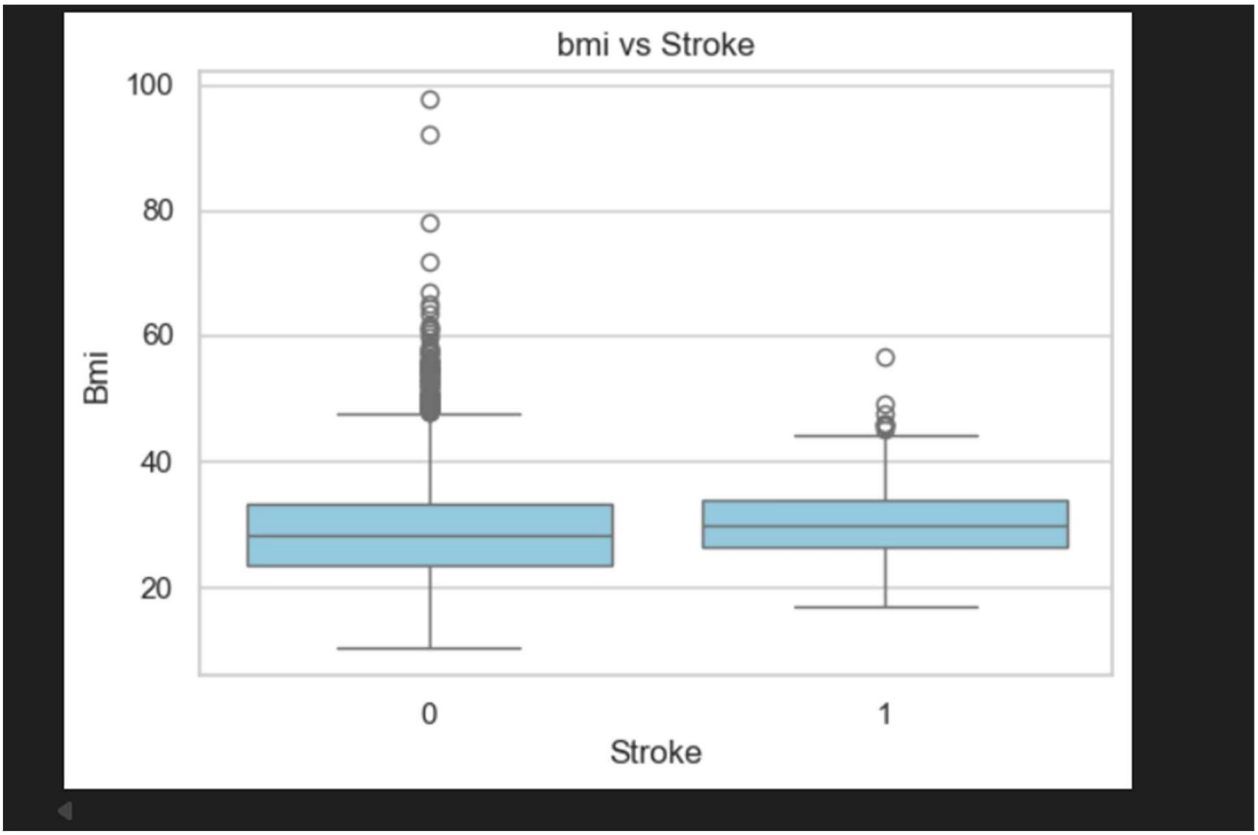

Distribution of Residence_type
Count
2500
2000
1500
1000
500
0
Urban
Rural
Residence_type

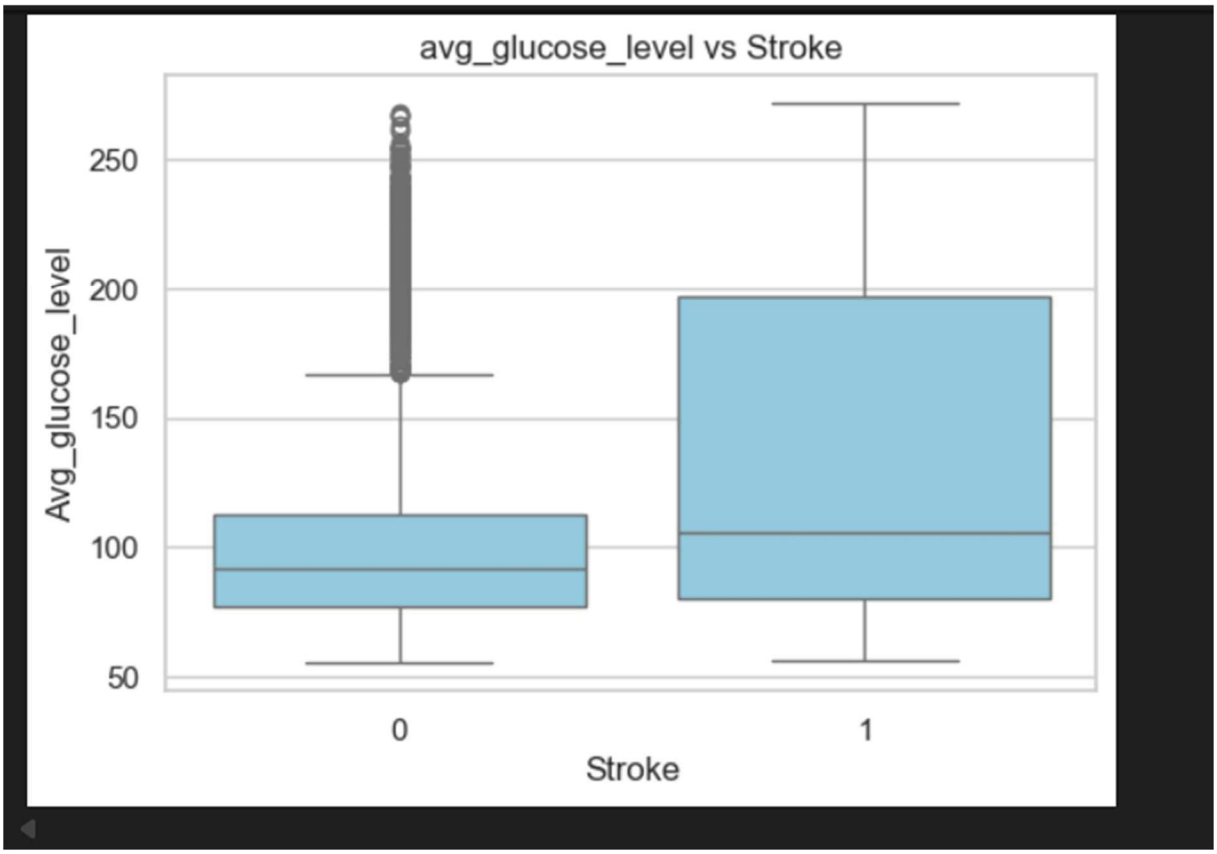





```
num_features = ['age', 'bmi', 'avg_glucose_level']
```

```
for col in num_features:
    plt.figure(figsize=(6,4))
    sns.boxplot(x='stroke', y=col, data=df, color='skyblue') # Use color instead
of palette
    plt.title(f'{col} vs Stroke')
    plt.xlabel('Stroke')
    plt.ylabel(col.capitalize())
    plt.show()

# Mean comparison table
print(df.groupby('stroke')[num_features].mean())
```


bmi vs Stroke
Bmi
100
80
60
40
20
0
1
Stroke


avg_glucose_level vs Stroke
Avg_glucose_level
250
200
150
100
50
0
1
Stroke

stroke			
0	41.971545	28.823064	104.795513
1	67.728193	30.471292	132.544739

INSIGHTS

old people have significantly higher risk of stroke

bmi doesnt seem to have large effect on stroke

people who have have higher glucose level tends to also have a stroke but need more analysis

Bivariate Analysis: Categorical Features vs Stroke

Bivariate analysis examines the relationship between **two variables**. In this section, we analyze how categorical features relate to the target variable (**stroke**) to identify potential risk factors and trends.

Code

```
cat_features = ['gender', 'hypertension', 'heart_disease', 'ever_married',
                'work_type', 'Residence_type', 'smoking_status']

for col in cat_features:
    cross_tab = pd.crosstab(df[col], df['stroke'], normalize='index') * 100
    cross_tab.plot(kind='bar', stacked=True, figsize=(6,4),
colormap='Paired')
    plt.title(f'Stroke by {col}')
    plt.ylabel('Percentage (%)')
    plt.xlabel(col.capitalize())
    plt.xticks(rotation=30)
    plt.legend(title='Stroke', labels=['No', 'Yes'])
    plt.show()
```

Code Explanation

1. Selecting categorical features

The list `cat_features` includes all relevant categorical predictors such as `gender`, `hypertension`, `work_type`, and `smoking_status`.

2. `pd.crosstab()`

- Creates a **cross-tabulation table** between the categorical feature and the target variable (`stroke`).

- `normalize='index'` converts counts to **row-wise percentages**, so each category shows the proportion of stroke and non-stroke cases.
- Multiplying by 100 converts proportions to **percentages**.

3. `cross_tab.plot(kind='bar', stacked=True, ...)`

- Plots a **stacked bar chart** showing the percentage of stroke vs non-stroke for each category.
- `colormap='Paired'` adds color distinction for better visualization.
- `stacked=True` allows easier comparison within each category.

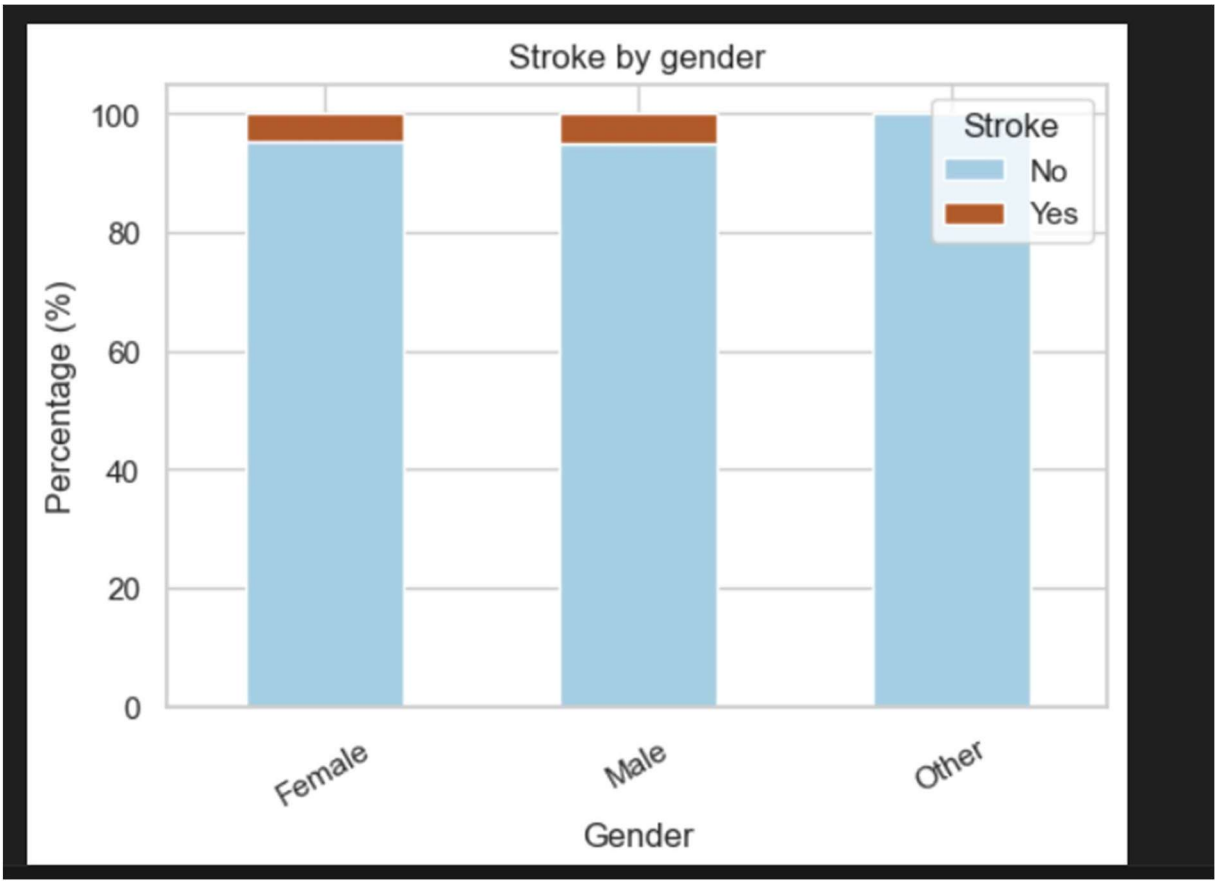
4. Labels and formatting

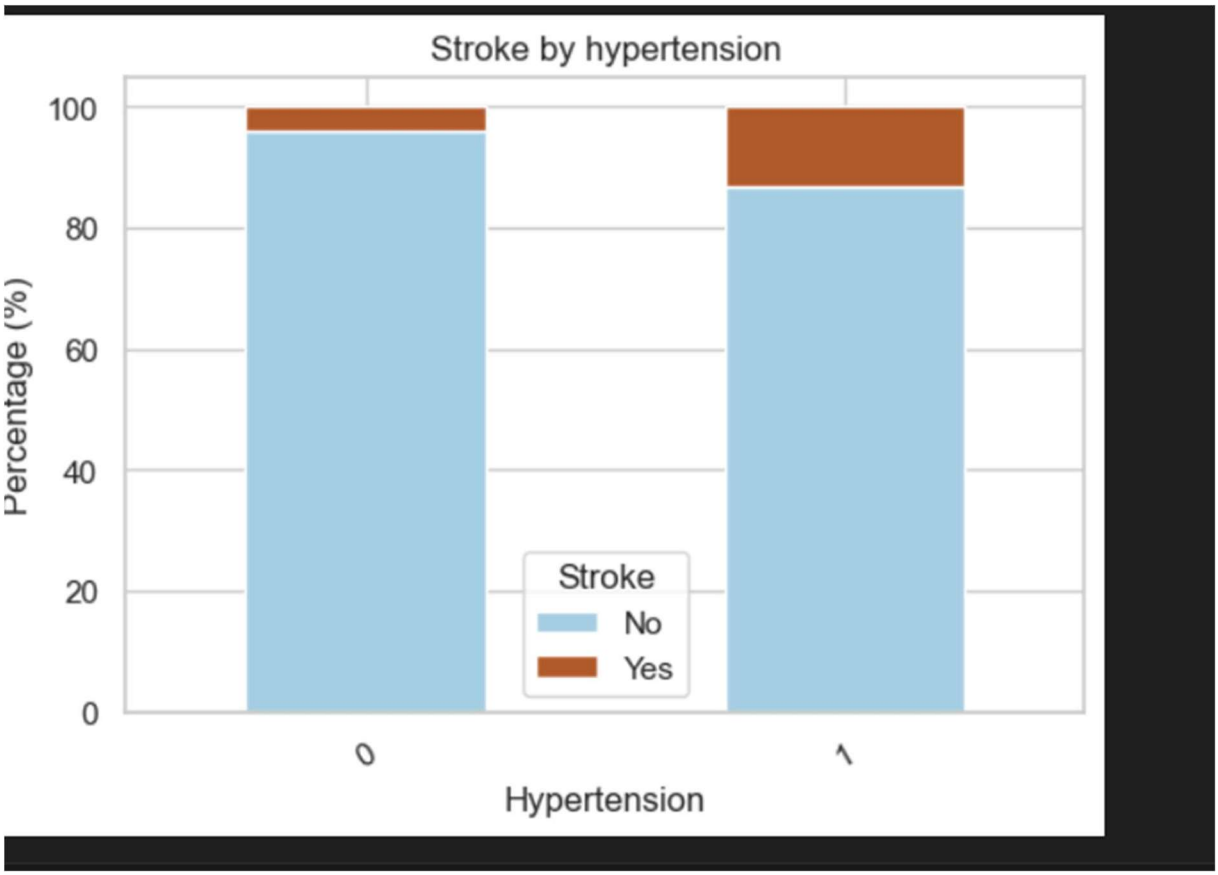
- Titles and axis labels make plots easy to interpret.
- Rotation of x-axis labels improves readability.
- Legend clearly shows which section represents stroke = 0 (No) and stroke = 1 (Yes).

Purpose of This Analysis

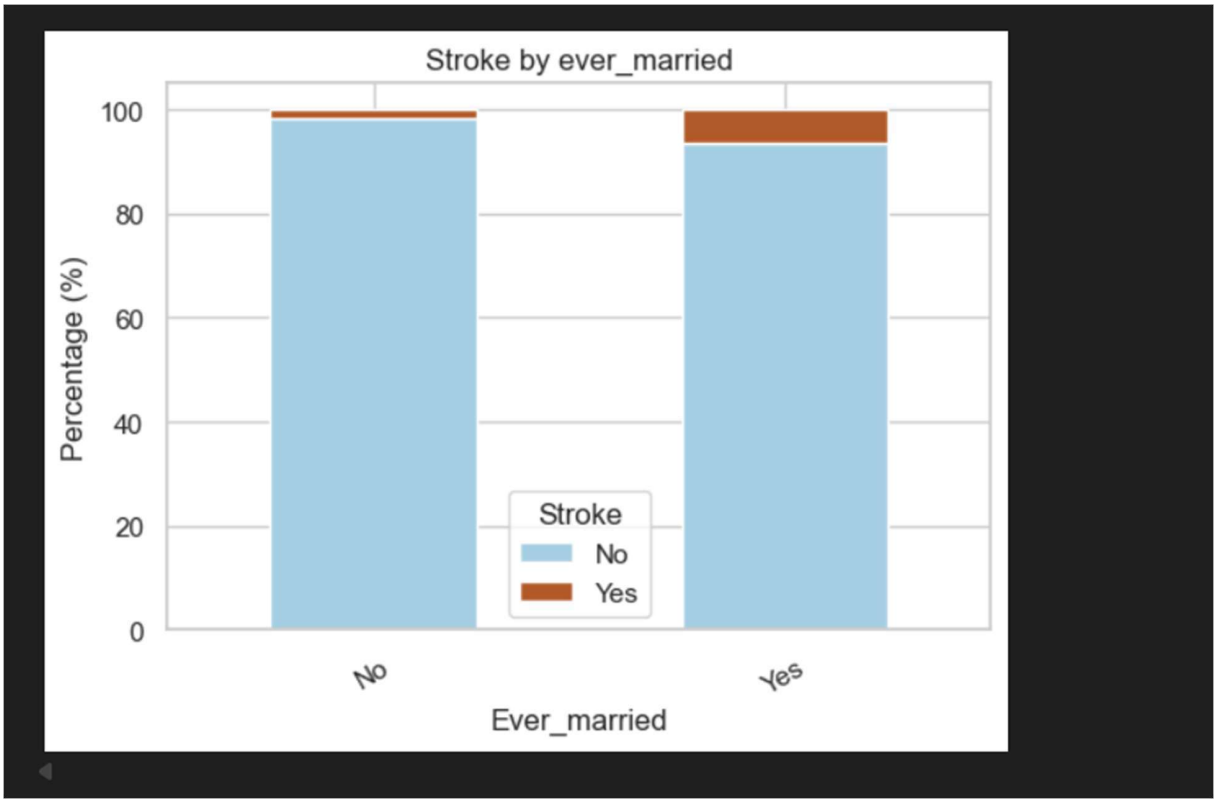
- To **visually assess** the impact of categorical features on the likelihood of stroke.
- To identify **high-risk categories** (e.g., people with hypertension or heart disease may have a higher percentage of stroke).
- Helps in **feature selection** and informs preprocessing decisions like encoding, grouping rare categories, or creating binary indicators.

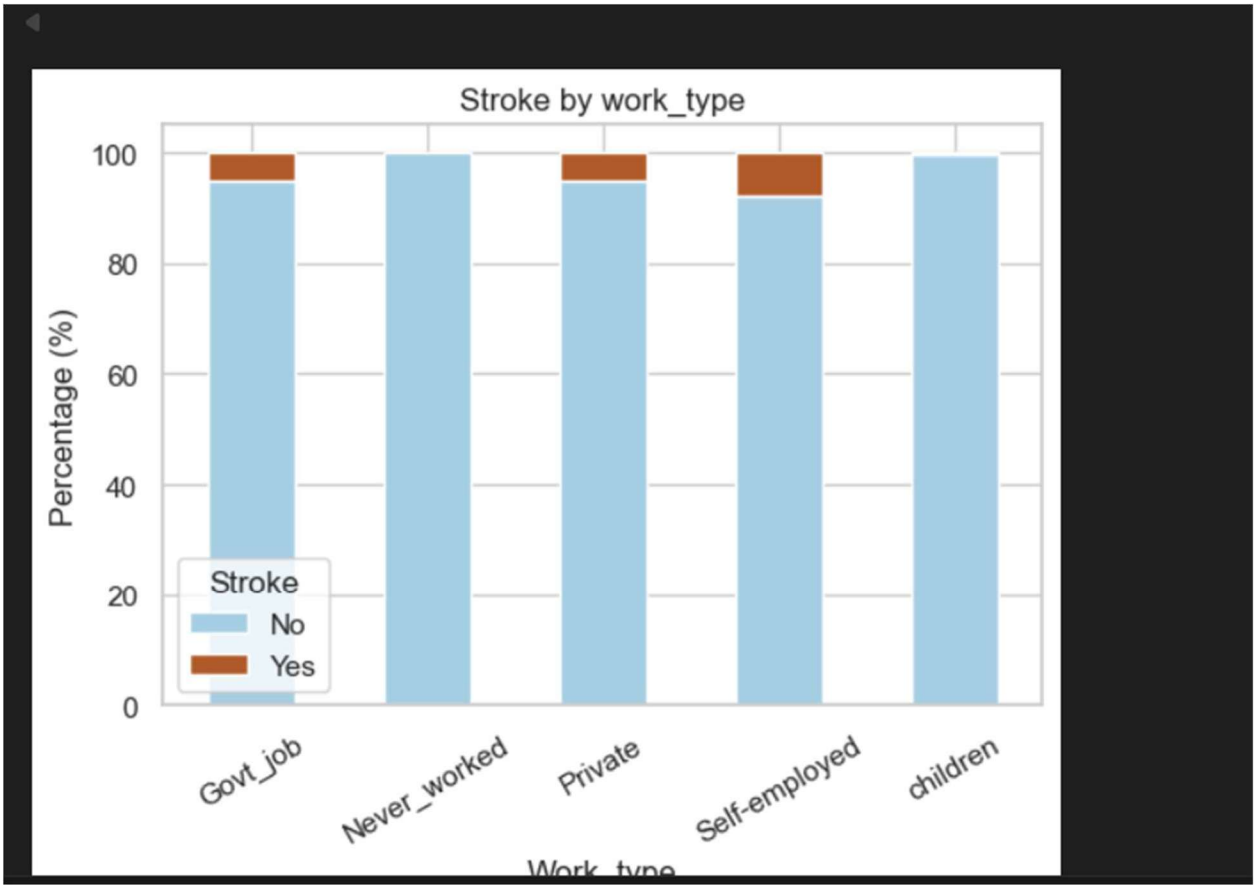
```
cat_features = ['gender', 'hypertension', 'heart_disease', 'ever_married',
                'work_type', 'Residence_type', 'smoking_status']

for col in cat_features:
    cross_tab = pd.crosstab(df[col], df['stroke'], normalize='index') *
100
    cross_tab.plot(kind='bar', stacked=True, figsize=(6,4),
colormap='Paired')
    plt.title(f'Stroke by {col}')
    plt.ylabel('Percentage (%)')
    plt.xlabel(col.capitalize())
    plt.xticks(rotation=30)
    plt.legend(title='Stroke', labels=['No', 'Yes'])
    plt.show()
```

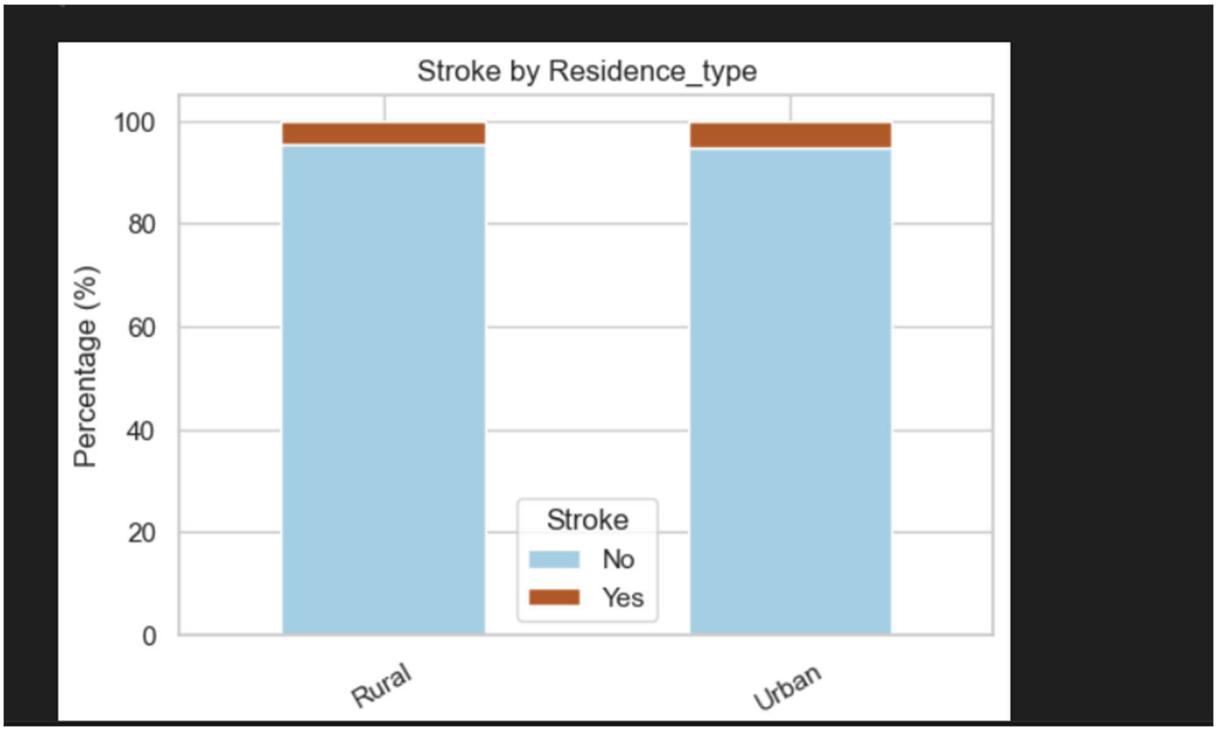

Stroke by gender
Stroke
No
Yes
Percentage (%)
100
80
60
40
20
0
Female
Male
Other
Gender

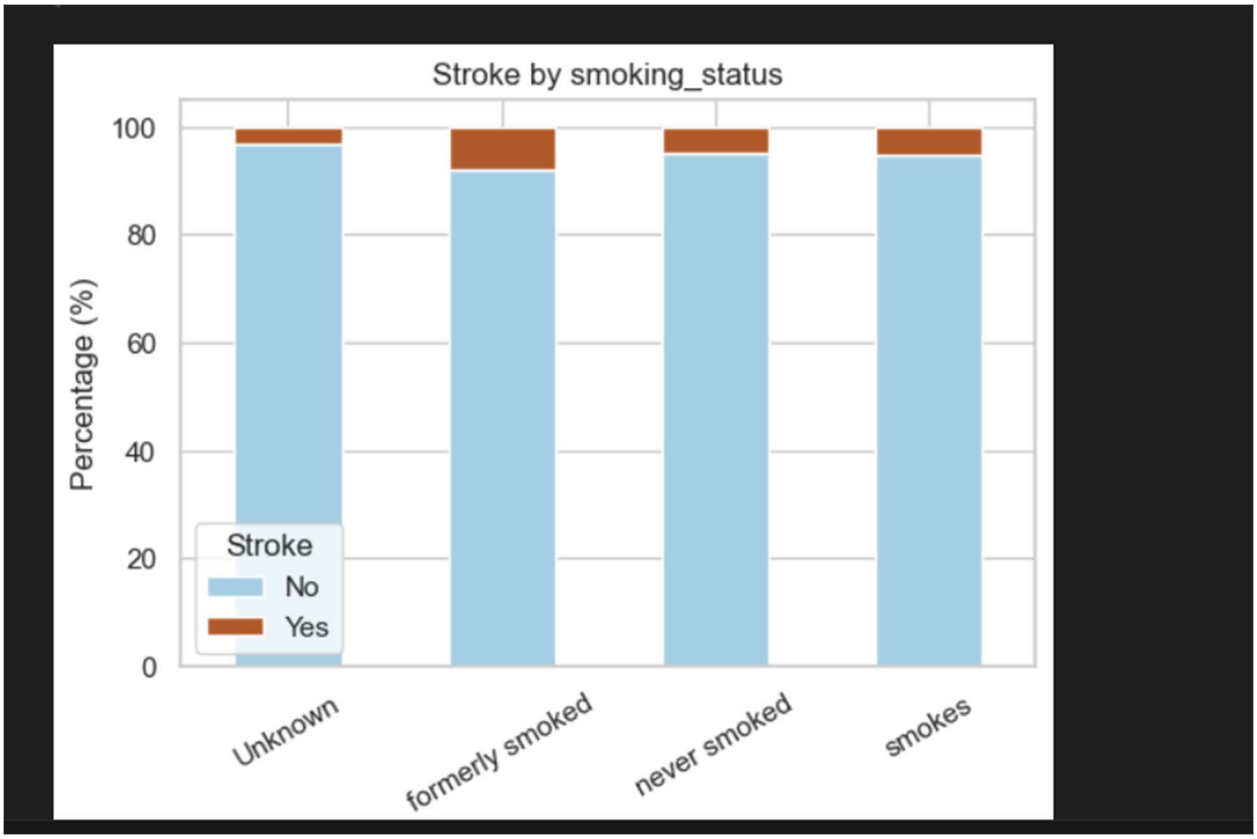

Stroke by hypertension
100
80
60
40
20
0
Percentage (%)
Stroke
No
Yes
0
1
Hypertension


Stroke by heart_disease
Percentage (%)
100
80
60
40
20
0
Stroke
No
Yes
0
1
Heart_disease


Stroke by ever_married
Percentage (%)
100
80
60
40
20
0
Stroke
No
Yes
No
Yes
Ever_married


Stroke by work_type
Percentage (%)
100
80
60
40
20
0
Stroke
No
Yes
Govt_job
Never_worked
Private
Self-employed
children
Work_type


Stroke by Residence_type
Percentage (%)
100
80
60
40
20
0
Stroke
No
Yes
Rural
Urban



INSIGHTS

Gender doesnt seem to have any direct effect with stroke

people with hpyertension have more stroke as it appears

people with heartdisesas also have more strokes

strangely marriage seems to have a relation with strokes as people who gets married have more stroke then single people this need more anlaysis

work type : people who work in government ,private or slef_employed have more chance of stroke than other work types

people living in urban have higher chance of haveing a stroke than people who live in rural but not significanlty hihger

frequent smoker have high chance of ahving a stroke

Correlation Analysis

Correlation analysis is a **multivariate analysis technique** that measures the strength and direction of the relationship between numerical variables. Understanding correlations helps identify features that are strongly associated with the target variable (**stroke**) and can guide feature selection for predictive modeling.

Code

```
# Compute correlation matrix for all numeric features
corr = df.corr(numeric_only=True)
```

```
# Visualize correlation using a heatmap
plt.figure(figsize=(8,6))
sns.heatmap(corr, annot=True, cmap='coolwarm', linewidths=0.5)
plt.title('Correlation Heatmap')
plt.show()

# Check correlation of each feature with the target
print(corr['stroke'].sort_values(ascending=False))

# Select numerical features including the target
num_features = ['age', 'bmi', 'avg_glucose_level', 'hypertension',
'heart_disease', 'stroke']

# Compute correlation matrix
corr_matrix = df[num_features].corr()

# Show correlation matrix
print(corr_matrix)

# Correlation of each numeric feature with the target
corr_with_target = corr_matrix['stroke'].sort_values(ascending=False)
print("Correlation with Stroke:\n", corr_with_target)
```

Code Explanation

1. `df.corr(numeric_only=True)`

- Computes the **Pearson correlation coefficient** between all numerical features.
- Values range from **-1 to +1**:
  - **+1** → perfect positive correlation
  - **-1** → perfect negative correlation
  - **0** → no linear correlation

2. Heatmap Visualization

```
sns.heatmap(corr, annot=True, cmap='coolwarm', linewidths=0.5)
```

- Displays the correlation matrix visually.
- **`annot=True`** shows numeric values in each cell.
- **`cmap='coolwarm'`** colors positive correlations in warm tones and negative in cool tones.
- **`linewidths=0.5`** adds grid lines for clarity.

3. Correlation with the Target

```
print(corr['stroke'].sort_values(ascending=False))
```

- Extracts the correlation of each feature specifically with **stroke**.
- Sorting helps quickly identify features with strong positive or negative associations.

4. Focused Numerical Features

```
num_features = ['age', 'bmi', 'avg_glucose_level', 'hypertension',
'heart_disease', 'stroke']
```

- Includes both continuous and binary numerical features.
- Computing correlations only among these ensures a clear comparison with the target.

5. Correlation Matrix & Target Correlation

```
corr_matrix = df[num_features].corr()
corr_with_target = corr_matrix['stroke'].sort_values(ascending=False)
```

- Shows correlations among selected numerical features.
- Identifies which variables have the **strongest association with stroke**, aiding in feature selection.

📌 Purpose of Correlation Analysis

- Detect linear relationships between numerical variables.
- Identify **predictor variables** most associated with stroke.
- Reveal **multicollinearity**, which can affect regression-based models.
- Guide **feature selection and engineering** to improve model performance.

```
corr = df.corr(numeric_only=True)
plt.figure(figsize=(8,6))
sns.heatmap(corr, annot=True, cmap='coolwarm', linewidths=0.5)
plt.title('Correlation Heatmap')
plt.show()

# Check correlation with target
print(corr['stroke'].sort_values(ascending=False))
```


```
stroke               1.000000
age                  0.245257
heart_disease        0.134914
avg_glucose_level    0.131945
hypertension         0.127904
bmi                  0.042374
id                   0.006388
Name: stroke, dtype: float64
```

```
# Select numerical features including the target
num_features = ['age', 'bmi', 'avg_glucose_level', 'hypertension', 'heart_disease', 'stroke']

# Compute correlation matrix
corr_matrix = df[num_features].corr()

# Show correlation matrix
print(corr_matrix)
```

0.0s

```
# Select numerical features including the target
num_features = ['age', 'bmi', 'avg_glucose_level', 'hypertension',
'heart_disease', 'stroke']

# Compute correlation matrix
corr_matrix = df[num_features].corr()

# Show correlation matrix
```

```
print(corr_matrix)
```

```
                       age       bmi  avg_glucose_level  hypertension  \
age               1.000000  0.333398           0.238171      0.276398
bmi               0.333398  1.000000           0.175502      0.167811
avg_glucose_level 0.238171  0.175502           1.000000      0.174474
hypertension      0.276398  0.167811           0.174474      1.000000
heart_disease     0.263796  0.041357           0.161857      0.108306
stroke            0.245257  0.042374           0.131945      0.127904

                   heart_disease    stroke
age                     0.263796  0.245257
bmi                     0.041357  0.042374
avg_glucose_level       0.161857  0.131945
hypertension            0.108306  0.127904
heart_disease           1.000000  0.134914
stroke                  0.134914  1.000000
```

```
# Correlation of each numeric feature with the target
corr_with_target = corr_matrix['stroke'].sort_values(ascending=False)
print("Correlation with Stroke:\n", corr_with_target)
```

```
Correlation with Stroke:
 stroke               1.000000
age                   0.245257
heart_disease         0.134914
avg_glucose_level     0.131945
hypertension          0.127904
bmi                   0.042374
Name: stroke, dtype: float64
```

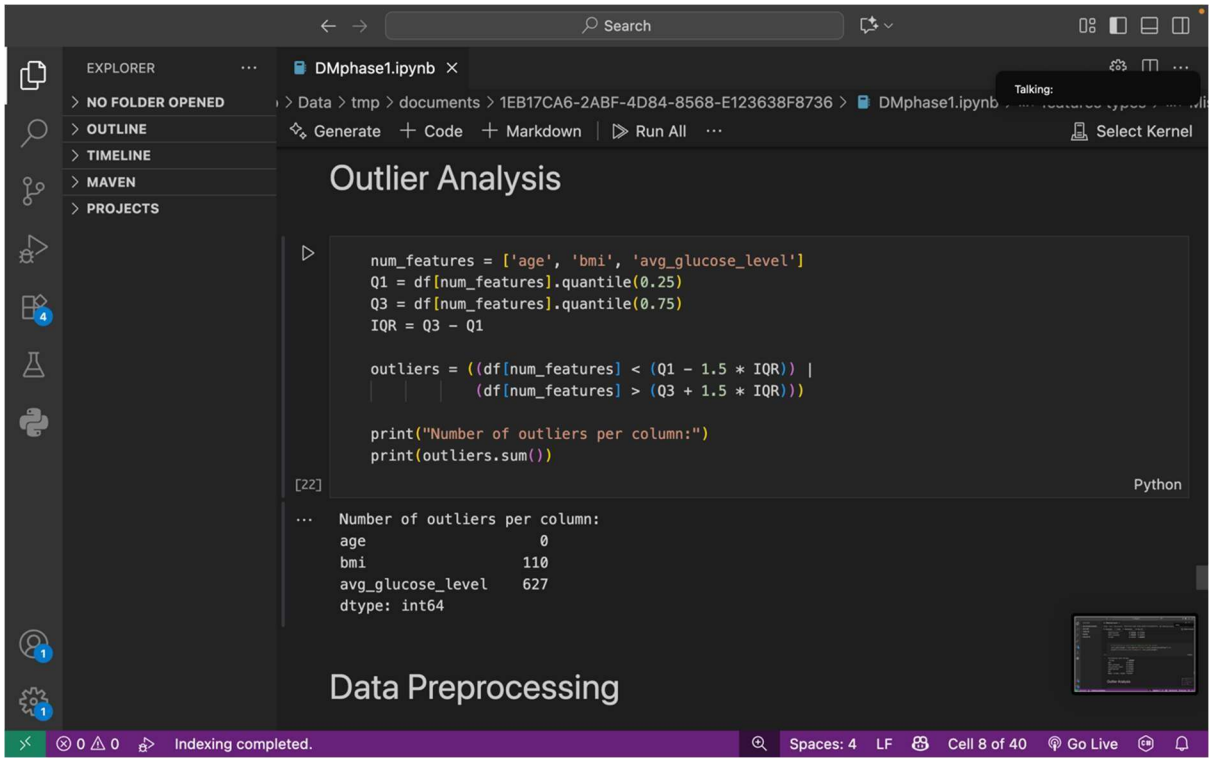
Outlier detection:

-takes selected numeric columns

-Computes Q1, Q3, and IQR

-Identifies outliers using the IQR rule

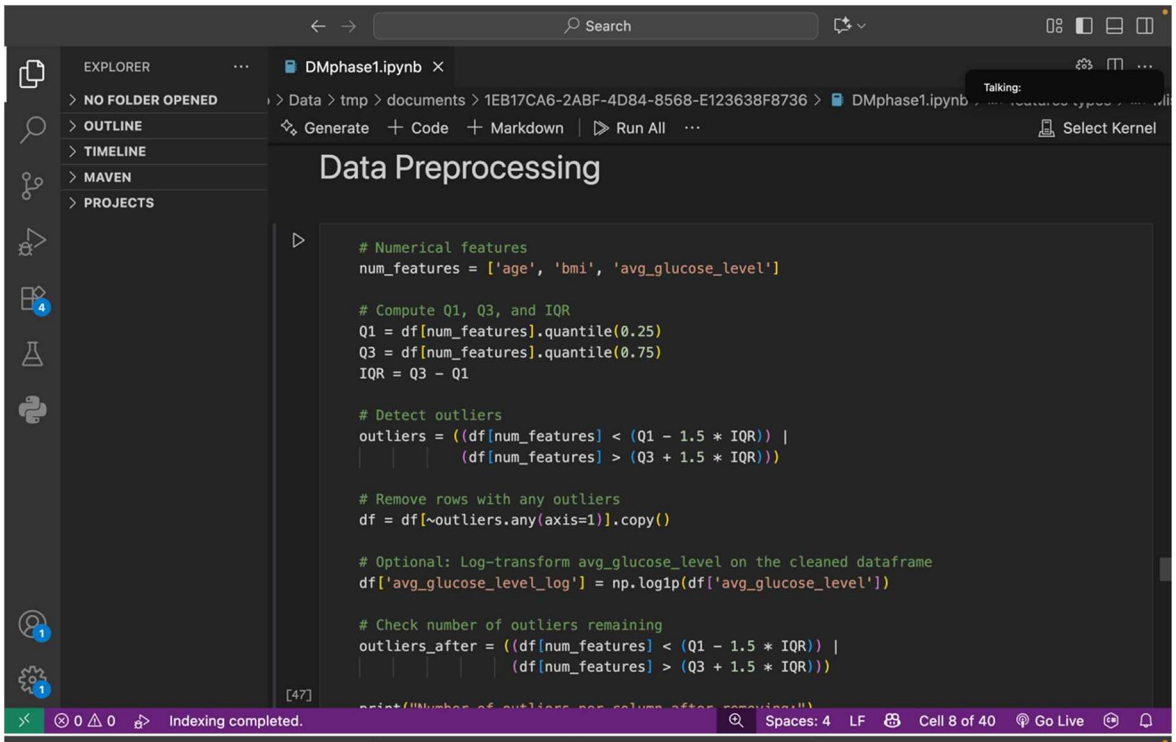
-Counts how many outliers appear in each column

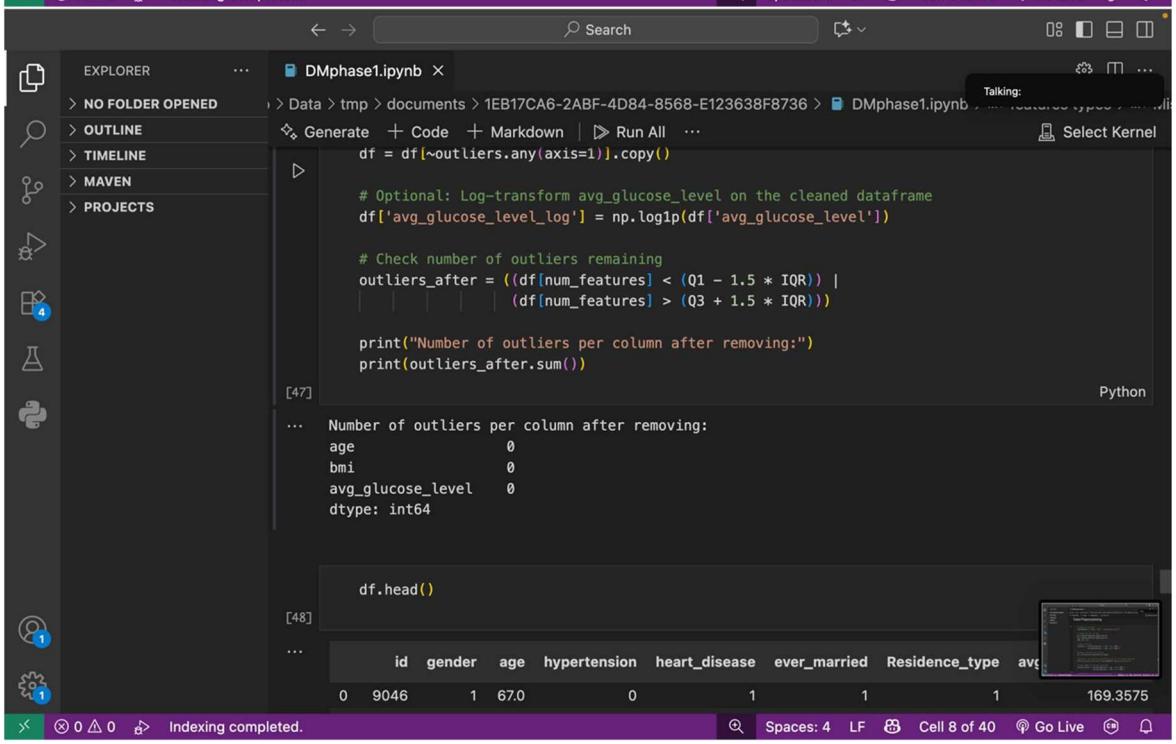


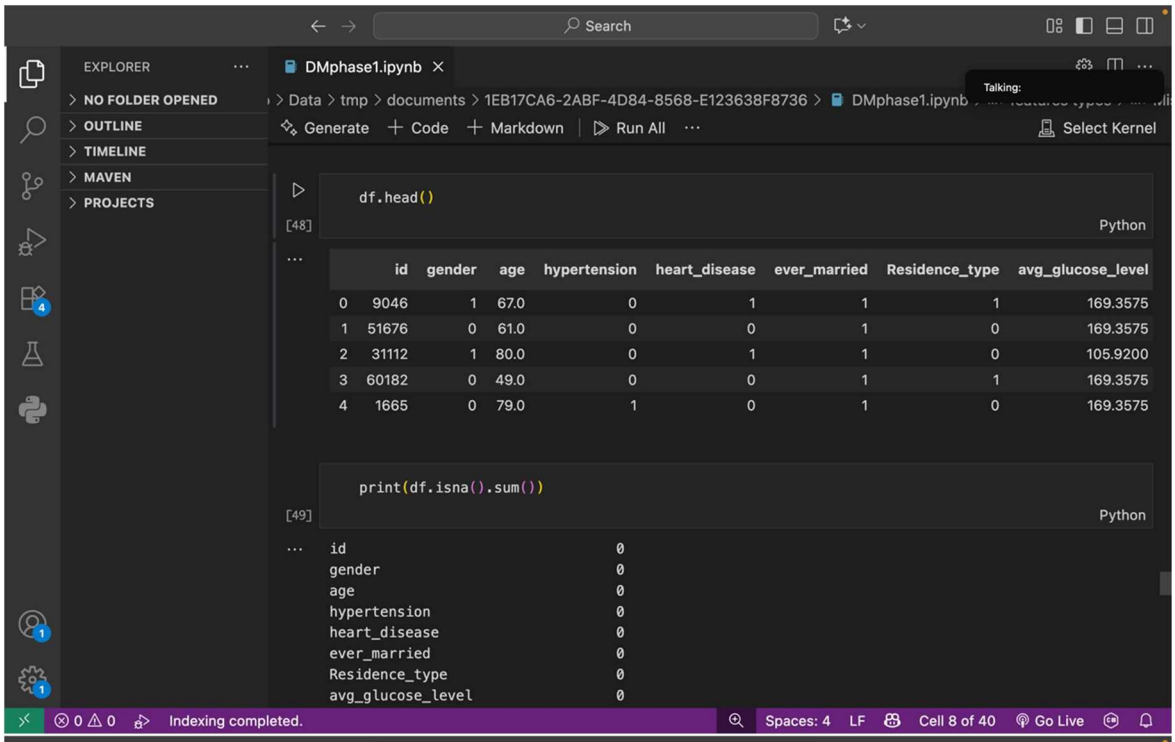
Data Processing:

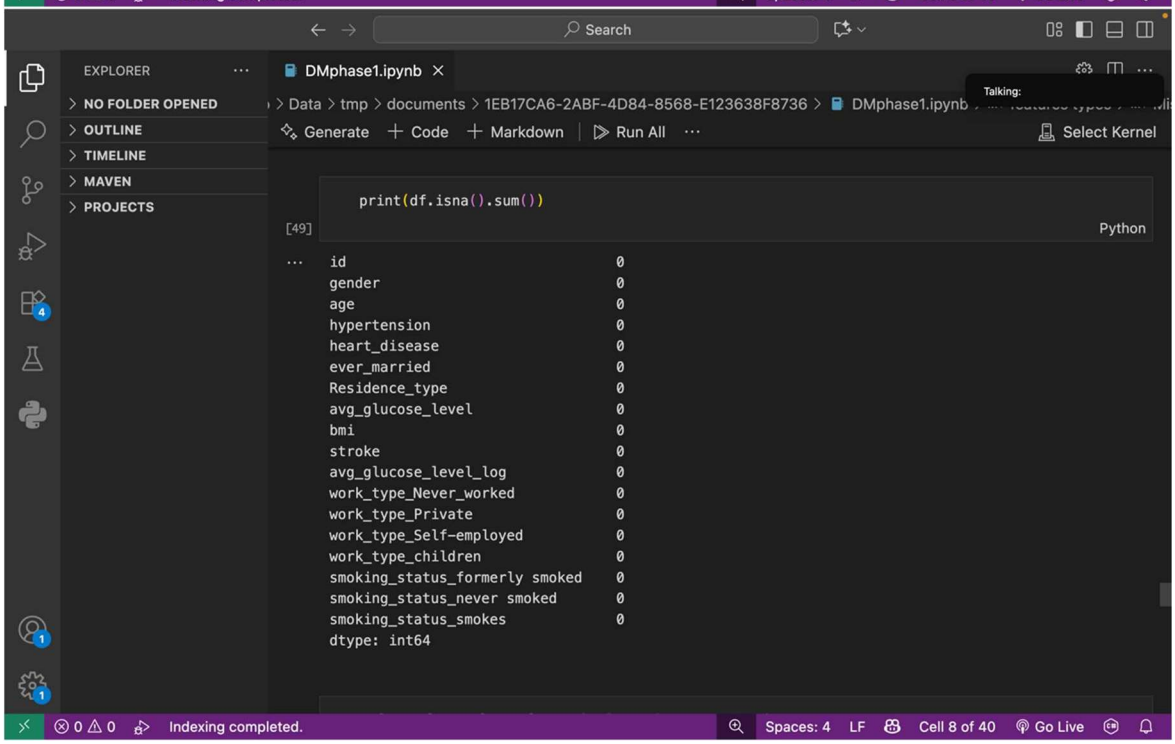
This code performs a full pre-processing pipeline for preparing a dataset for modelling. It begins by selecting the numerical columns (*age*, *bmi*, and *avg_glucose_level*) and identifying outliers using the IQR method by computing the 25th percentile (Q1), 75th percentile (Q3), and the interquartile range (IQR). Any rows containing values outside the range **Q1 – 1.5×IQR to Q3 + 1.5×IQR** are removed. After cleaning, a log transformation is applied to *avg_glucose_level* to reduce skewness. The code then fixes invalid ages by replacing any age below 1 with NaN and later filling missing age values with the median. Next, it handles categorical variables by filling missing values—using the mode when available or `"Unknown"` if the entire column is missing—then encoding binary categorical variables with Label Encoder and one-hot encoding multi-class categorical columns like *work type* and *smoking status*. Missing numeric values such as BMI are

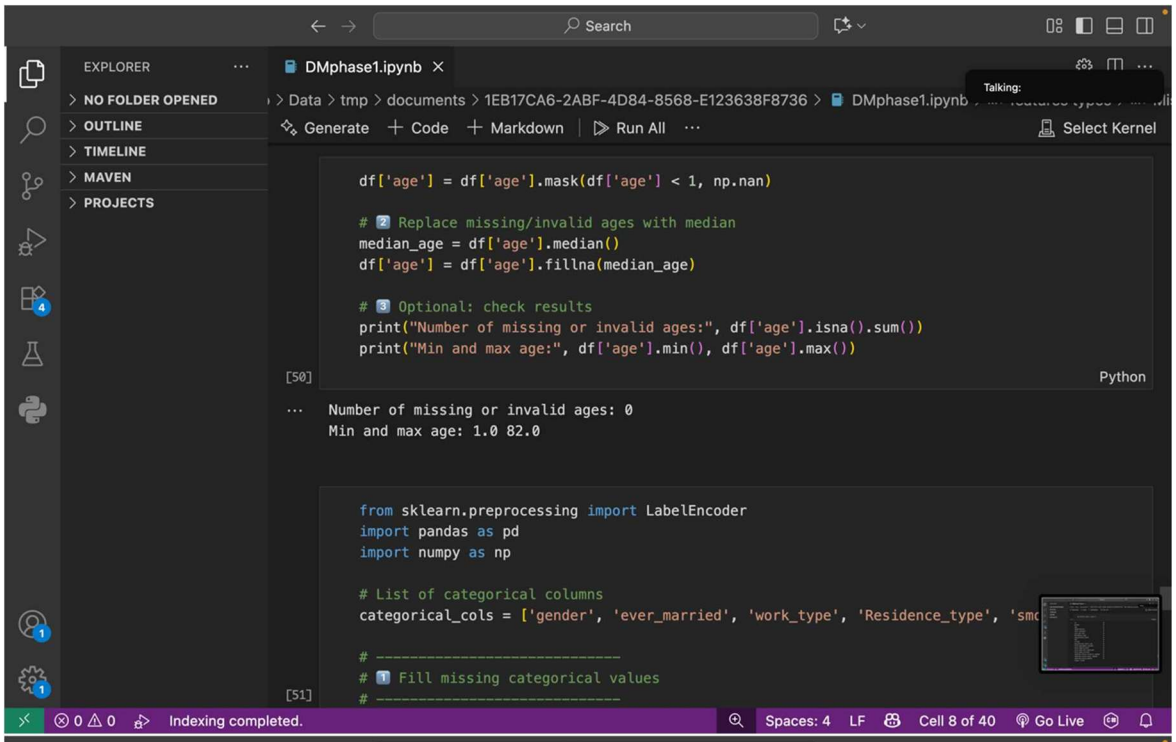
also filled with their median to maintain consistency. After pre-processing, the code prints missing values, data types, and a preview of the cleaned dataset. Finally, it visualizes the distribution of numerical features using histograms and calculates stroke rates for different work types and smoking statuses by computing the proportion of stroke cases within each encoded category, displaying these results in bar plots.

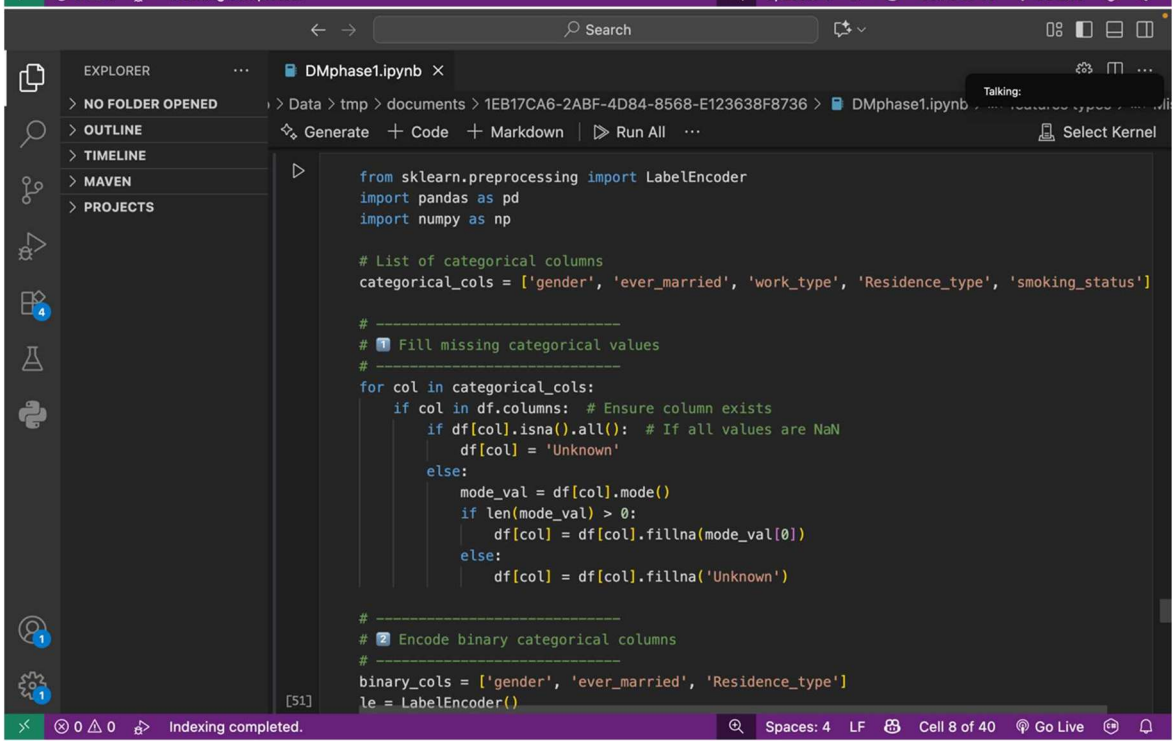

Search
EXPLORER
DMphase1.ipynb
NO FOLDER OPENED
OUTLINE
TIMELINE
MAVEN
PROJECTS
Data > tmp > documents > 1EB17CA6-2ABF-4D84-8568-E123638F8736 > DMphase1.ipynb
Talking:
Generate
Code
Markdown
Run All
Select Kernel
Data Preprocessing
# Numerical features
num_features = ['age', 'bmi', 'avg_glucose_level']
# Compute Q1, Q3, and IQR
Q1 = df[num_features].quantile(0.25)
Q3 = df[num_features].quantile(0.75)
IQR = Q3 - Q1
# Detect outliers
outliers = ((df[num_features] < (Q1 - 1.5 * IQR)) |
(df[num_features] > (Q3 + 1.5 * IQR)))
# Remove rows with any outliers
df = df[~outliers.any(axis=1)].copy()
# Optional: Log-transform avg_glucose_level on the cleaned dataframe
df['avg_glucose_level_log'] = np.log1p(df['avg_glucose_level'])
# Check number of outliers remaining
outliers_after = ((df[num_features] < (Q1 - 1.5 * IQR)) |
(df[num_features] > (Q3 + 1.5 * IQR)))
[47]
Indexing completed.
Spaces: 4
LF
Cell 8 of 40
Go Live

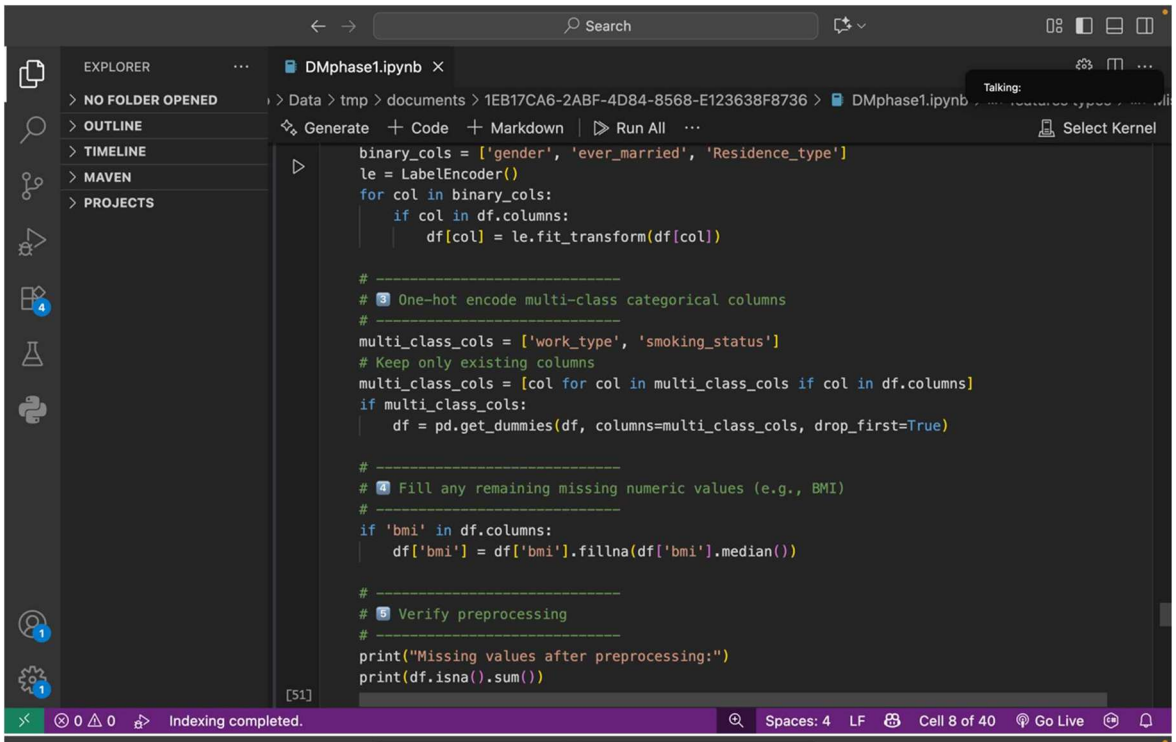

Search
EXPLORER
DMphase1.ipynb
NO FOLDER OPENED
OUTLINE
TIMELINE
MAVEN
PROJECTS
Data > tmp > documents > 1EB17CA6-2ABF-4D84-8568-E123638F8736 > DMphase1.ipynb
Talking:
Generate
Code
Markdown
Run All
Select Kernel
df = df[~outliers.any(axis=1)].copy()
# Optional: Log-transform avg_glucose_level on the cleaned dataframe
df['avg_glucose_level_log'] = np.log1p(df['avg_glucose_level'])
# Check number of outliers remaining
outliers_after = ((df[num_features] < (Q1 - 1.5 * IQR)) |
(df[num_features] > (Q3 + 1.5 * IQR)))
print("Number of outliers per column after removing:")
print(outliers_after.sum())
[47]
Python
Number of outliers per column after removing:
age 0
bmi 0
avg_glucose_level 0
dtype: int64
df.head()
[48]
id gender age hypertension heart_disease ever_married Residence_type
0 9046 1 67.0 0 1 1 1 169.3575
Indexing completed.
Spaces: 4
LF
Cell 8 of 40
Go Live

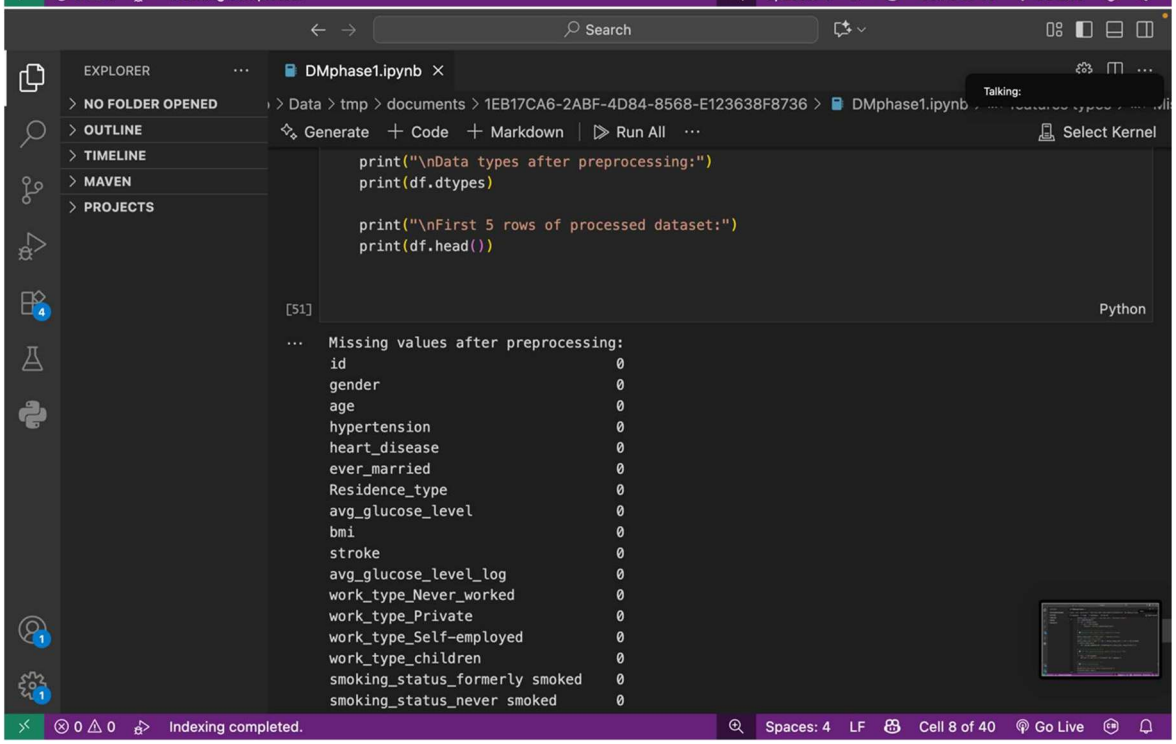

Search
EXPLORER
DMphase1.ipynb
NO FOLDER OPENED
OUTLINE
TIMELINE
MAVEN
PROJECTS
Data > tmp > documents > 1EB17CA6-2ABF-4D84-8568-E123638F8736 > DMphase1.ipynb
Talking:
Generate
Code
Markdown
Run All
Select Kernel
df.head()
[48]
Python
id gender age hypertension heart_disease ever_married Residence_type avg_glucose_level
0 9046 1 67.0 0 1 1 1 169.3575
1 51676 0 61.0 0 0 1 0 169.3575
2 31112 1 80.0 0 1 1 0 105.9200
3 60182 0 49.0 0 0 1 1 169.3575
4 1665 0 79.0 1 0 1 0 169.3575
print(df.isna().sum())
[49]
Python
id 0
gender 0
age 0
hypertension 0
heart_disease 0
ever_married 0
Residence_type 0
avg_glucose_level 0
0 0
Indexing completed.
Spaces: 4
LF
Cell 8 of 40
Go Live

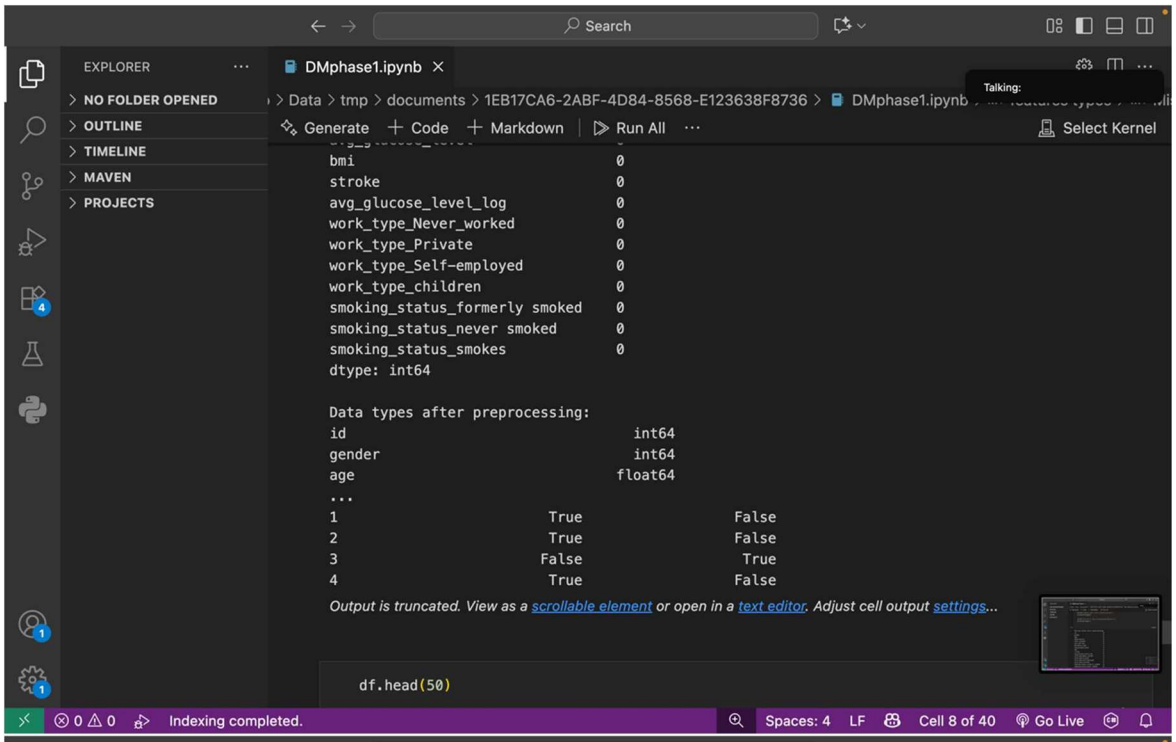

Search
EXPLORER
DMphase1.ipynb
NO FOLDER OPENED
OUTLINE
TIMELINE
MAVEN
PROJECTS
Data > tmp > documents > 1EB17CA6-2ABF-4D84-8568-E123638F8736 > DMphase1.ipynb
Talking:
Generate
Code
Markdown
Run All
Select Kernel
print(df.isna().sum())
[49]
Python
id 0
gender 0
age 0
hypertension 0
heart_disease 0
ever_married 0
Residence_type 0
avg_glucose_level 0
bmi 0
stroke 0
avg_glucose_level_log 0
work_type_Never_worked 0
work_type_Private 0
work_type_Self-employed 0
work_type_children 0
smoking_status_formerly smoked 0
smoking_status_never smoked 0
smoking_status_smokes 0
dtype: int64
0 0
Indexing completed.
Spaces: 4
LF
Cell 8 of 40
Go Live

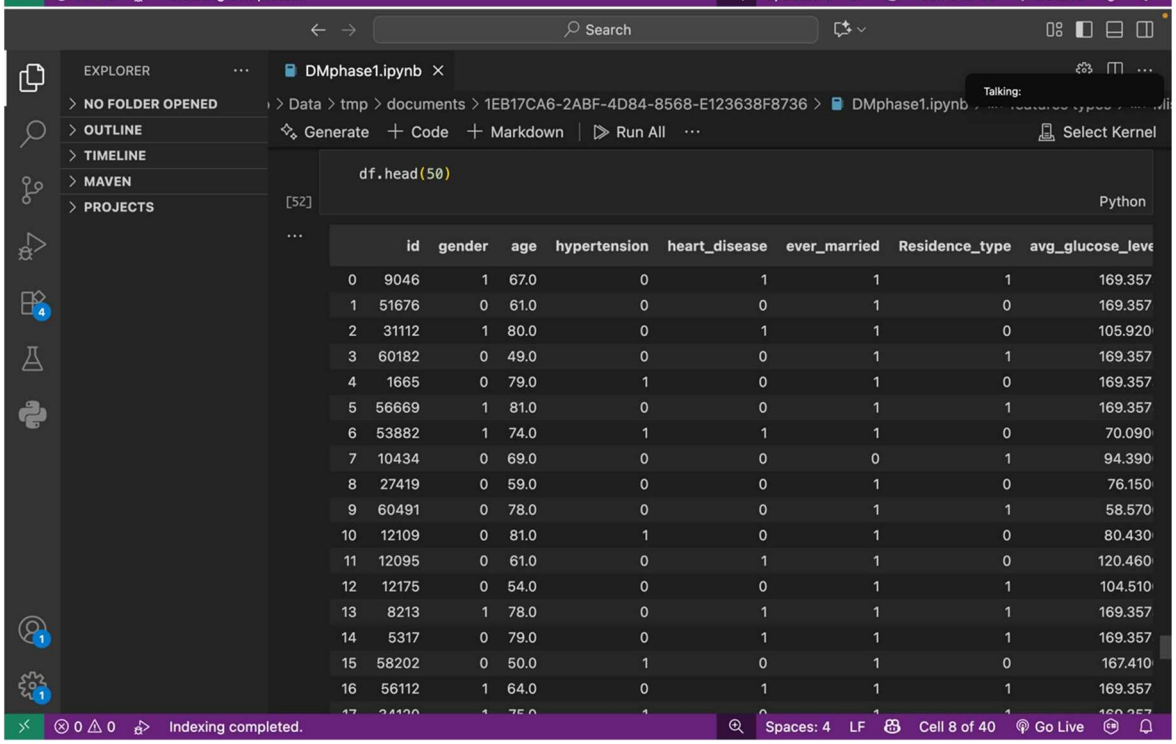

Search
EXPLORER
DMphase1.ipynb
NO FOLDER OPENED
OUTLINE
TIMELINE
MAVEN
PROJECTS
Talking:
Data > tmp > documents > 1EB17CA6-2ABF-4D84-8568-E123638F8736 > DMphase1.ipynb
Generate
Code
Markdown
Run All
Select Kernel
df['age'] = df['age'].mask(df['age'] < 1, np.nan)
# 2 Replace missing/invalid ages with median
median_age = df['age'].median()
df['age'] = df['age'].fillna(median_age)
# 3 Optional: check results
print("Number of missing or invalid ages:", df['age'].isna().sum())
print("Min and max age:", df['age'].min(), df['age'].max())
[50]
Python
Number of missing or invalid ages: 0
Min and max age: 1.0 82.0
from sklearn.preprocessing import LabelEncoder
import pandas as pd
import numpy as np
# List of categorical columns
categorical_cols = ['gender', 'ever_married', 'work_type', 'Residence_type', 'smo
# ------------------------------
# 1 Fill missing categorical values
# ------------------------------
[51]
0 0
Indexing completed.
Spaces: 4
LF
Cell 8 of 40
Go Live


Search
EXPLORER
DMphase1.ipynb
NO FOLDER OPENED
OUTLINE
TIMELINE
MAVEN
PROJECTS
Talking:
Data > tmp > documents > 1EB17CA6-2ABF-4D84-8568-E123638F8736 > DMphase1.ipynb
Generate
Code
Markdown
Run All
Select Kernel
from sklearn.preprocessing import LabelEncoder
import pandas as pd
import numpy as np
# List of categorical columns
categorical_cols = ['gender', 'ever_married', 'work_type', 'Residence_type', 'smoking_status']
# ------------------------------
# 1 Fill missing categorical values
# ------------------------------
for col in categorical_cols:
if col in df.columns: # Ensure column exists
if df[col].isna().all(): # If all values are NaN
df[col] = 'Unknown'
else:
mode_val = df[col].mode()
if len(mode_val) > 0:
df[col] = df[col].fillna(mode_val[0])
else:
df[col] = df[col].fillna('Unknown')
# ------------------------------
# 2 Encode binary categorical columns
# ------------------------------
binary_cols = ['gender', 'ever_married', 'Residence_type']
[51]
le = LabelEncoder()
0 0
Indexing completed.
Spaces: 4
LF
Cell 8 of 40
Go Live


Search
EXPLORER
DMphase1.ipynb
NO FOLDER OPENED
OUTLINE
TIMELINE
MAVEN
PROJECTS
Data > tmp > documents > 1EB17CA6-2ABF-4D84-8568-E123638F8736 > DMphase1.ipynb
Talking:
Generate
Code
Markdown
Run All
Select Kernel
binary_cols = ['gender', 'ever_married', 'Residence_type']
le = LabelEncoder()
for col in binary_cols:
if col in df.columns:
df[col] = le.fit_transform(df[col])
# ------------------------------
# 3 One-hot encode multi-class categorical columns
# ------------------------------
multi_class_cols = ['work_type', 'smoking_status']
# Keep only existing columns
multi_class_cols = [col for col in multi_class_cols if col in df.columns]
if multi_class_cols:
df = pd.get_dummies(df, columns=multi_class_cols, drop_first=True)
# ------------------------------
# 4 Fill any remaining missing numeric values (e.g., BMI)
# ------------------------------
if 'bmi' in df.columns:
df['bmi'] = df['bmi'].fillna(df['bmi'].median())
# ------------------------------
# 5 Verify preprocessing
# ------------------------------
print("Missing values after preprocessing:")
print(df.isna().sum())
[51]
0 0 0
Indexing completed.
Spaces: 4
LF
Cell 8 of 40
Go Live


Search
EXPLORER
DMphase1.ipynb
NO FOLDER OPENED
OUTLINE
TIMELINE
MAVEN
PROJECTS
Data > tmp > documents > 1EB17CA6-2ABF-4D84-8568-E123638F8736 > DMphase1.ipynb
Talking:
Generate
Code
Markdown
Run All
Select Kernel
print("\nData types after preprocessing:")
print(df.dtypes)
print("\nFirst 5 rows of processed dataset:")
print(df.head())
[51]
Python
Missing values after preprocessing:
id 0
gender 0
age 0
hypertension 0
heart_disease 0
ever_married 0
Residence_type 0
avg_glucose_level 0
bmi 0
stroke 0
avg_glucose_level_log 0
work_type_Never_worked 0
work_type_Private 0
work_type_Self-employed 0
work_type_children 0
smoking_status_formerly smoked 0
smoking_status_never smoked 0
0 0 0
Indexing completed.
Spaces: 4
LF
Cell 8 of 40
Go Live


Search
EXPLORER
DMphase1.ipynb
NO FOLDER OPENED
OUTLINE
TIMELINE
MAVEN
PROJECTS
Data > tmp > documents > 1EB17CA6-2ABF-4D84-8568-E123638F8736 > DMphase1.ipynb
Talking:
Generate
Code
Markdown
Run All
Select Kernel
bmi 0
stroke 0
avg_glucose_level_log 0
work_type_Never_worked 0
work_type_Private 0
work_type_Self-employed 0
work_type_children 0
smoking_status_formerly smoked 0
smoking_status_never smoked 0
smoking_status_smokes 0
dtype: int64
Data types after preprocessing:
id int64
gender int64
age float64
...
1 True False
2 True False
3 False True
4 True False
Output is truncated. View as a scrollable element or open in a text editor. Adjust cell output settings...
df.head(50)
0 0 Indexing completed.
Spaces: 4 LF Cell 8 of 40 Go Live


Search
EXPLORER
DMphase1.ipynb
NO FOLDER OPENED
OUTLINE
TIMELINE
MAVEN
PROJECTS
Data > tmp > documents > 1EB17CA6-2ABF-4D84-8568-E123638F8736 > DMphase1.ipynb
Talking:
Generate
Code
Markdown
Run All
Select Kernel
df.head(50)
[52]
Python
id gender age hypertension heart_disease ever_married Residence_type avg_glucose_leve
0 9046 1 67.0 0 1 1 1 169.357
1 51676 0 61.0 0 0 1 0 169.357
2 31112 1 80.0 0 1 1 0 105.920
3 60182 0 49.0 0 0 1 1 169.357
4 1665 0 79.0 1 0 1 0 169.357
5 56669 1 81.0 0 0 1 1 169.357
6 53882 1 74.0 1 1 1 0 70.090
7 10434 0 69.0 0 0 0 1 94.390
8 27419 0 59.0 0 0 1 0 76.150
9 60491 0 78.0 0 0 1 1 58.570
10 12109 0 81.0 1 0 1 0 80.430
11 12095 0 61.0 0 1 1 0 120.460
12 12175 0 54.0 0 0 1 1 104.510
13 8213 1 78.0 0 1 1 1 169.357
14 5317 0 79.0 0 1 1 1 169.357
15 58202 0 50.0 1 0 1 0 167.410
16 56112 1 64.0 0 1 1 1 169.357
0 0 Indexing completed.
Spaces: 4 LF Cell 8 of 40 Go Live

Search

EXPLORER

DMphase1.ipynb

NO FOLDER OPENED

OUTLINE

TIMELINE

MAVEN

PROJECTS

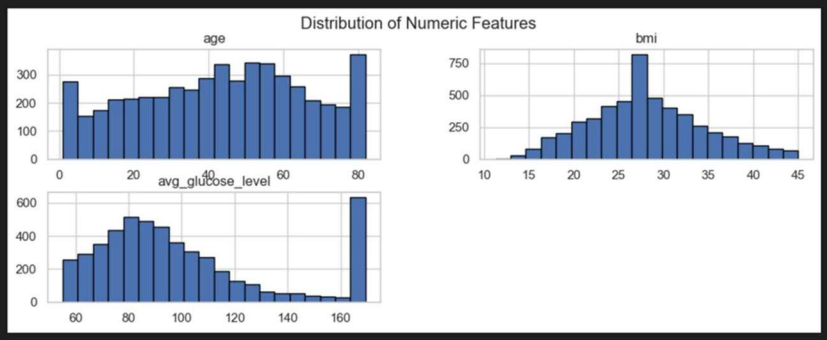
Data > tmp > documents > 1EB17CA6-2ABF-4D84-8568-E123638F8736 > DMphase1.ipynb

Talking:

Generate + Code + Markdown | Run All ... Select Kernel

```
numeric_cols = ['age', 'bmi', 'avg_glucose_level']

df[numeric_cols].hist(figsize=(12, 4), bins=20, edgecolor='black')
plt.suptitle("Distribution of Numeric Features")
plt.show()
```

[53] Python



0 0 Indexing completed.

Spaces: 4 LF Cell 8 of 40 Go Live

Search

EXPLORER

DMphase1.ipynb

NO FOLDER OPENED

OUTLINE

TIMELINE

MAVEN

PROJECTS

Data > tmp > documents > 1EB17CA6-2ABF-4D84-8568-E123638F8736 > DMphase1.ipynb

Talking:

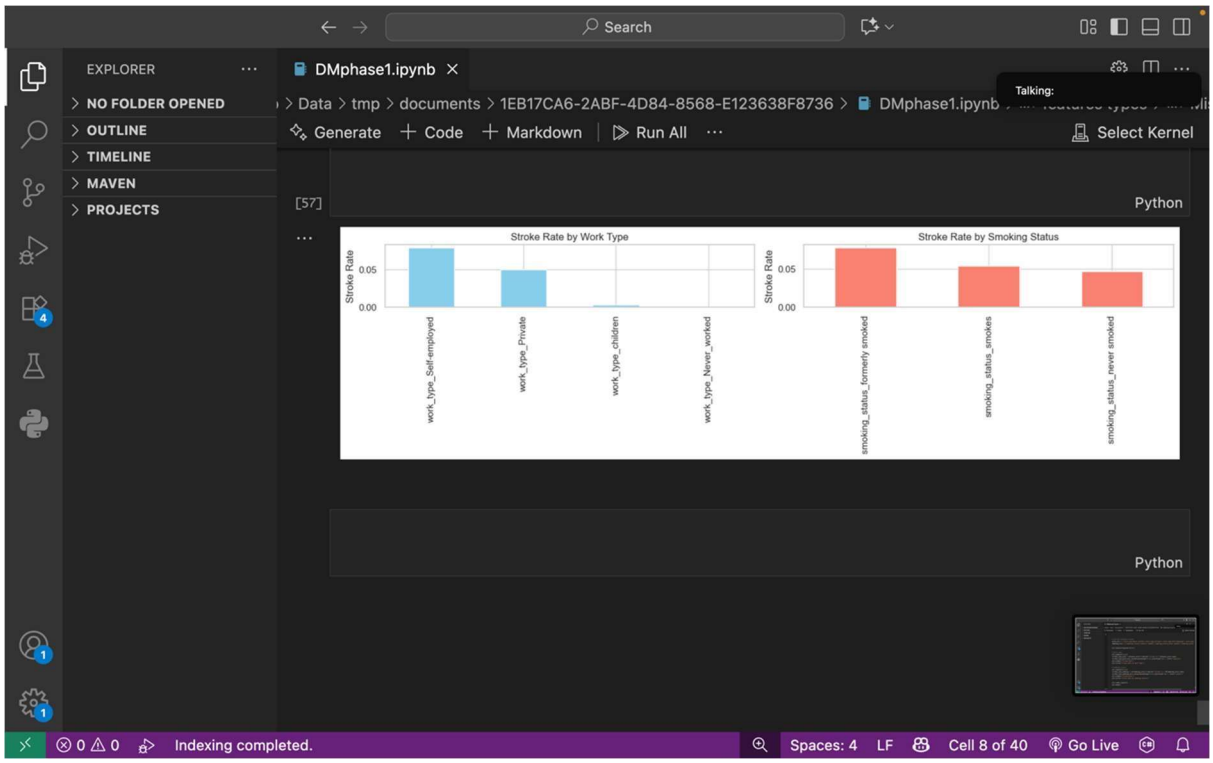
Generate + Code + Markdown | Run All ... Select Kernel

```
# One-hot encoded columns
work_cols = ['work_type_Never_worked','work_type_Private','work_type_Self-employed','work_type
smoking_cols = ['smoking_status_formerly smoked','smoking_status_never smoked','smoking_status

plt.figure(figsize=(14,4))

# Work type
plt.subplot(1,2,1)
stroke_rate_work = df[work_cols].T.dot(df['stroke']) / df[work_cols].sum()
stroke_rate_work.sort_values(ascending=False).plot(kind='bar', color='skyblue')
plt.ylabel('Stroke Rate')
plt.title('Stroke Rate by Work Type')

# Smoking status
plt.subplot(1,2,2)
stroke_rate_smoking = df[smoking_cols].T.dot(df['stroke']) / df[smoking_cols].sum()
stroke_rate_smoking.sort_values(ascending=False).plot(kind='bar', color='salmon')
plt.ylabel('Stroke Rate')
plt.title('Stroke Rate by Smoking Status')

plt.tight_layout()
plt.show()
```

[57]

0 0 Indexing completed.

Spaces: 4 LF Cell 8 of 40 Go Live



Train-Test Split / Dataset Preparation

Insight:

- Before training the model, the dataset is split into **features (x)** and **target (y)**.
  - Features include all relevant patient data except the `id` (irrelevant for prediction) and the target variable `stroke`.
- The **train-test split** is performed to evaluate model performance on unseen data.
  - `test_size=0.2` means **20% of the data** is reserved for testing, while **80% is used for training**.
  - `stratify=y` ensures that the proportion of stroke vs non-stroke cases is preserved in both training and testing sets, which is important because stroke occurrences are relatively rare (imbalanced dataset).
- Setting a `random_state` ensures reproducibility of the results.
- The printed shapes confirm the sizes of the training and testing sets, which allows understanding how many samples the model will learn from and how many will be used to evaluate its performance.

Optional Visualization / Table:

Dataset	Samples	Features
Training	80%	n
Testing	20%	n

- This split is a crucial step to **prevent overfitting** and to get a realistic measure of the model's predictive ability on new patients.

```
from sklearn.model_selection import train_test_split

# Drop 'id' and target column 'stroke' from features
X = df.drop(['id', 'stroke'], axis=1)
```

```
y = df['stroke']

X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42, stratify=y
)

print("Training set shape:", X_train.shape)
print("Testing set shape:", X_test.shape)
# Model Selection
Note: Label Class is imbalanced we may need to use SMOTE
```

Modeling and Evaluation Libraries

Insight to write:

- This block imports all the **necessary libraries for building, tuning, and evaluating classification models** for stroke prediction.

1. **Modeling:**
   - `RandomForestClassifier` – an ensemble model that combines multiple decision trees to improve accuracy and reduce overfitting.
   - `LogisticRegression` – a baseline model for binary classification, interpretable and effective for understanding feature influence.
2. **Data Preprocessing:**
   - `StandardScaler` – standardizes features by removing the mean and scaling to unit variance, important for models like Logistic Regression.
   - `Pipeline` – allows chaining preprocessing and modeling steps together, ensuring reproducibility and cleaner code.
3. **Hyperparameter Tuning:**
   - `GridSearchCV` – automatically searches for the best combination of hyperparameters using cross-validation to optimize model performance.
4. **Evaluation Metrics:**
   - `accuracy_score` – proportion of correctly predicted instances.
   - `precision_score` – fraction of predicted positives that are actually positive.
   - `recall_score` – fraction of actual positives correctly identified.
   - `f1_score` – harmonic mean of precision and recall, especially useful for imbalanced datasets.
   - `confusion_matrix` – summarizes prediction results, showing true vs predicted class counts.
   - `roc_curve` & `roc_auc_score` – used to evaluate the model's ability to distinguish between classes across different thresholds.

```
from sklearn.metrics import roc_curve, roc_auc_score
from sklearn.metrics import accuracy_score, precision_score, recall_score,
f1_score,confusion_matrix
from sklearn.pipeline import Pipeline
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import GridSearchCV
from sklearn.linear_model import LogisticRegression
from sklearn.preprocessing import StandardScaler

```

Custom Evaluation Function for Classification

Insight:

- This function `evaluate_classification` is designed to **thoroughly assess the performance of the stroke prediction model**. It combines visualizations and metrics in one place for easier interpretation.

1. **Confusion Matrix:**
   - Plots a **heatmap** showing true vs predicted classes.
   - Displays both counts and percentages, making it easier to understand how well the model predicts each class (stroke vs no stroke).
2. **Overall Metrics:**
   - Calculates **accuracy**, giving the proportion of correctly classified instances.
   - Shows **per-class metrics** including precision, recall, and F1-score, which are particularly important for **imbalanced datasets** like stroke prediction where positive cases are rare.
3. **ROC-AUC Evaluation (Optional):**
   - If predicted probabilities are provided, the function plots the **ROC curve** and calculates the **AUC score**, measuring the model's ability to discriminate between classes across all thresholds.

```
def evaluate_classification(y_true, y_pred, y_proba=None, labels=None, figsize=(6,5)):
    """
    Plots a labeled confusion matrix as a heatmap with counts and percentages,
    prints precision, recall, F1 for each class, overall accuracy,
    and optionally plots ROC curve.

    y_true  : true labels
    y_pred  : predicted labels
    y_proba : predicted probabilities for positive class (for ROC-AUC)
    labels  : list of class labels (e.g., [0,1])
    figsize : size of the confusion matrix plot
    """

    if labels is None:
        labels = [0, 1]

    # ---- Confusion Matrix ----
    cm = confusion_matrix(y_true, y_pred, labels=labels)
    cm_sum = cm.sum()
    cm_perc = cm / cm_sum * 100

    # Create a DataFrame with counts and percentages
    annot = np.empty_like(cm).astype(str)
    nrows, ncols = cm.shape
    for i in range(nrows):
        for j in range(ncols):
            annot[i, j] = f"{cm[i, j]}\n({cm_perc[i, j]:.1f}%)"

    plt.figure(figsize=figsize)
    sns.heatmap(cm, annot=annot, fmt='', cmap='Blues', cbar=False,
                xticklabels=labels, yticklabels=labels)
    plt.ylabel('Actual', fontsize=12)
```

```
    plt.xlabel('Predicted', fontsize=12)
    plt.title('Confusion Matrix', fontsize=14)
    plt.show()

    # ---- Metrics ----
    metrics_overall = {
        "Accuracy": accuracy_score(y_true, y_pred)
    }

    print("Overall Metrics:\n")
    display(pd.DataFrame(metrics_overall, index=["Score"]).T)

    # Per-class precision, recall, f1
    precision = precision_score(y_true, y_pred, labels=labels, average=None,
zero_division=0)
    recall = recall_score(y_true, y_pred, labels=labels, average=None, zero_division=0)
    f1 = f1_score(y_true, y_pred, labels=labels, average=None, zero_division=0)

    class_metrics_df = pd.DataFrame({
        "Precision": precision,
        "Recall": recall,
        "F1-Score": f1
    }, index=labels)

    print("Per-Class Metrics:\n")
    display(class_metrics_df)

    # ---- ROC-AUC ----
    if y_proba is not None:
        auc_score = roc_auc_score(y_true, y_proba)
        fpr, tpr, _ = roc_curve(y_true, y_proba)
        plt.figure(figsize=(6,5))
        plt.plot(fpr, tpr, label=f"ROC Curve (AUC = {auc_score:.3f})")
        plt.plot([0,1], [0,1], linestyle='--', color='gray')
        plt.xlabel("False Positive Rate")
        plt.ylabel("True Positive Rate")
        plt.title("ROC Curve", fontsize=14)
        plt.legend()
        plt.grid(True)
        plt.show()
        print(f"ROC-AUC Score: {auc_score:.4f}")

```

Random Forest Model Training and Evaluation

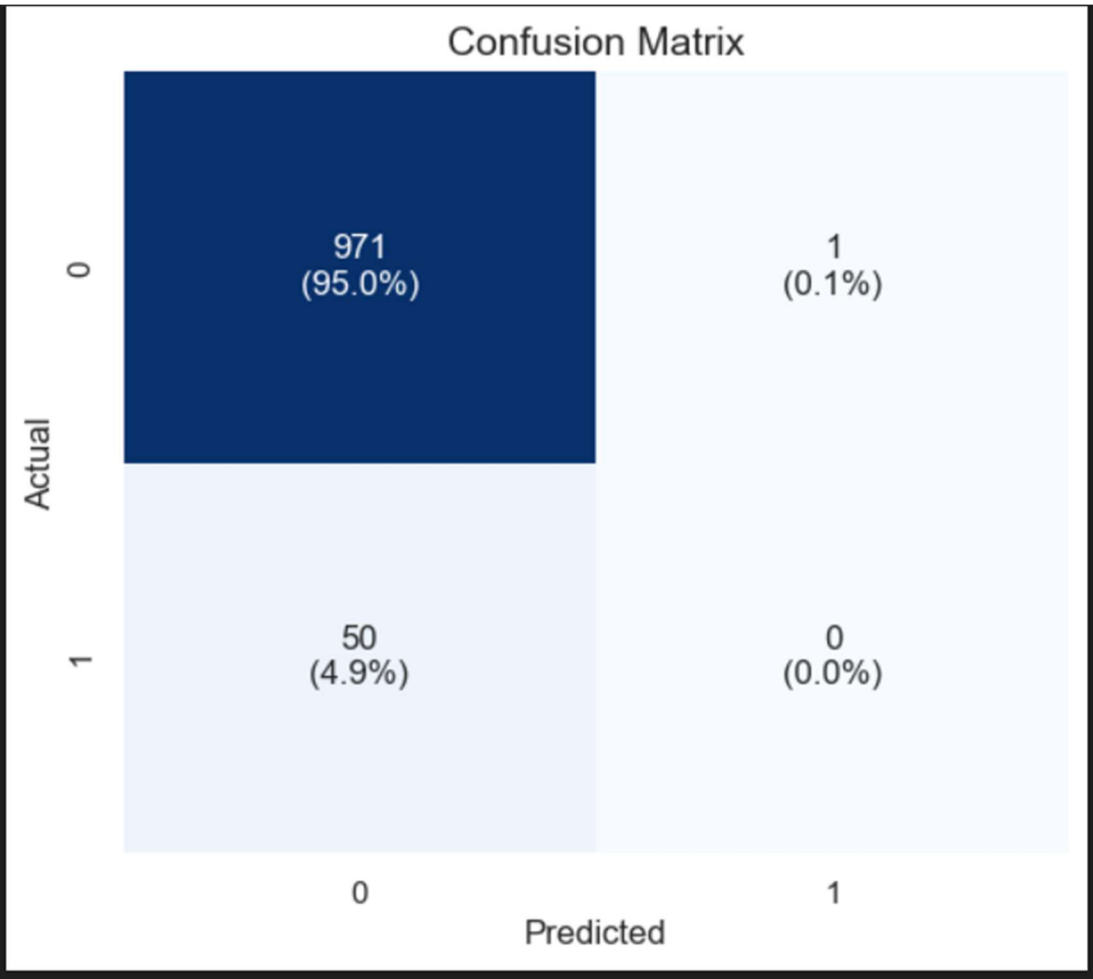
Insight to write:

1. **Model Training:**
    - A **Random Forest Classifier** with 100 trees (`n_estimators=100`) is trained on the training dataset.

- Random Forest is chosen because it is **robust, handles non-linear relationships well, and can manage feature interactions** without requiring extensive preprocessing.

2. **Predictions:**
   - The model generates both **class predictions (`y_pred`)** and **probabilities for the positive class (`y_proba`)**.
   - Probability outputs are useful for threshold adjustment and ROC-AUC evaluation.
3. **Evaluation:**
   - The custom `evaluate_classification` function is used to assess model performance.
   - **Outputs include:**
     - **Confusion matrix** with counts and percentages for actual vs predicted classes.
     - **Overall accuracy** and **per-class metrics** (precision, recall, F1-score), which are crucial given the dataset is **imbalanced** (stroke cases are rare).
     - **ROC curve and AUC score**, indicating how well the model can discriminate between patients likely to get a stroke and those who are not.

```
# ---- Train Random Forest ----
rf_model = RandomForestClassifier(n_estimators=100, random_state=42)
rf_model.fit(X_train, y_train)

# ---- Predictions ----
y_pred = rf_model.predict(X_test)
y_proba = rf_model.predict_proba(X_test)[:, 1]  # probability for positive class

# ---- Evaluate using pretty function ----
evaluate_classification(y_test, y_pred, y_proba, labels=[0,1])  # change labels if
needed
```



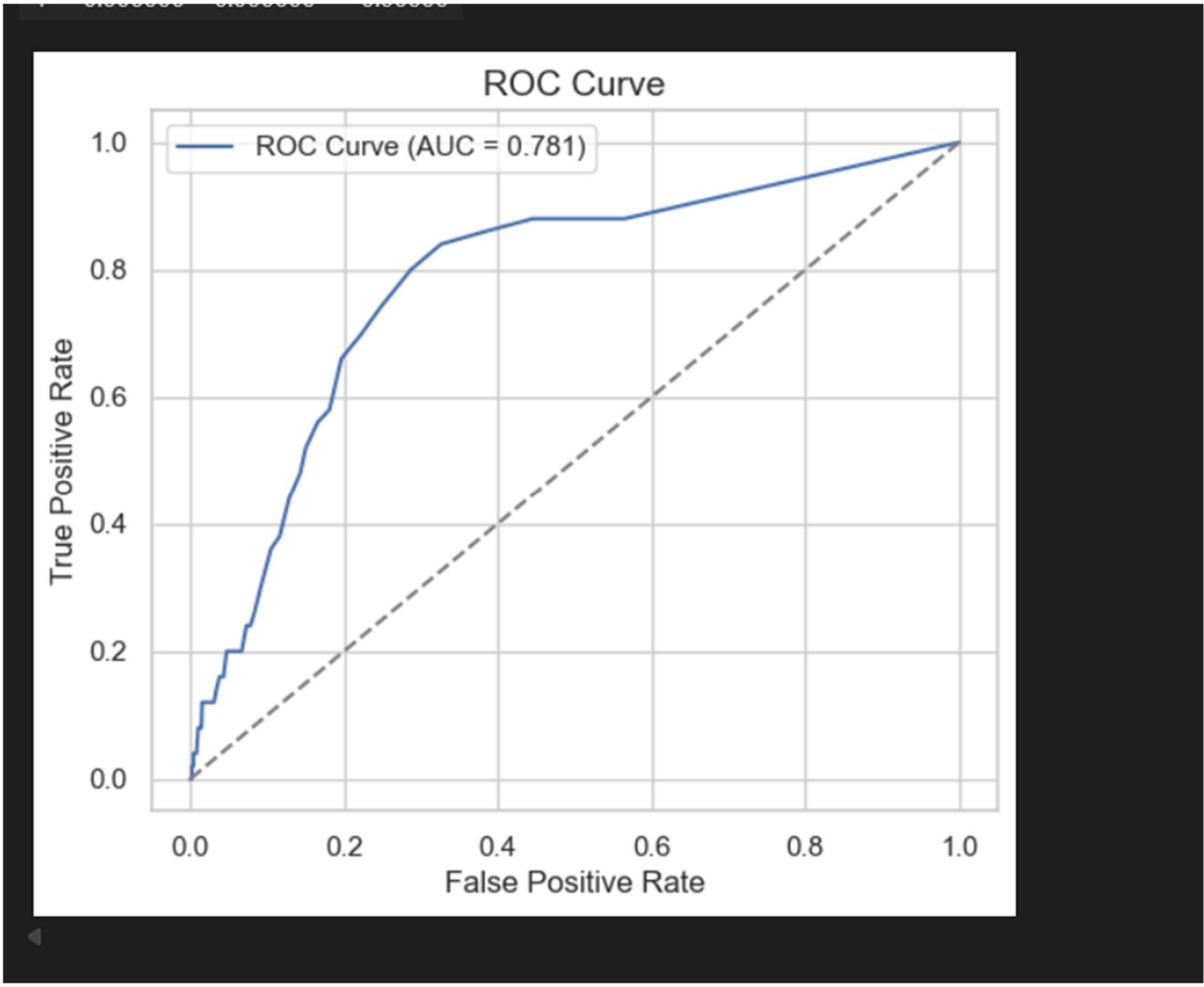
Overall Metrics:

Overall Metrics:

	Score
Accuracy	0.950098

Per-Class Metrics:

	Precision	Recall	F1-Score
0	0.951028	0.998971	0.97441
1	0.000000	0.000000	0.00000



Random Forest with Class Balancing

Insight to write:

1. **Handling Class Imbalance:**
   - The `class_weight='balanced'` parameter adjusts the weight of each class inversely proportional to its frequency.
   - This is crucial for the stroke dataset because **positive cases (stroke) are much rarer than negative cases**, and without balancing, the model might ignore them.
2. **Model Training:**
   - A Random Forest with 100 trees is trained on the **balanced dataset**, improving its ability to detect the minority class.
3. **Predictions:**

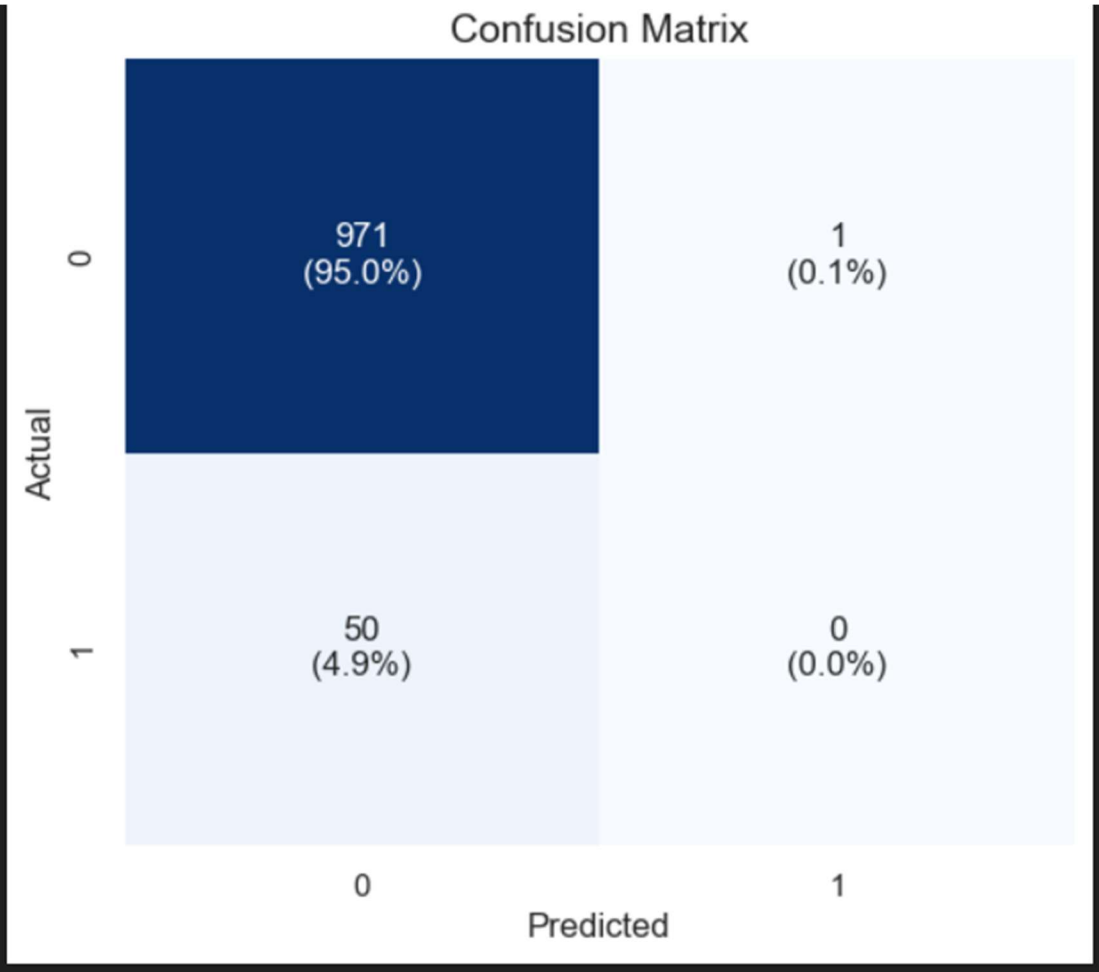
- The model generates both **class predictions (`y_pred`)** and **probabilities for stroke (`y_proba`)**, enabling threshold tuning and ROC analysis.

4. **Evaluation:**
   - The custom `evaluate_classification` function is used to display:
     - Confusion matrix (counts and percentages)
     - Overall accuracy and per-class metrics (precision, recall, F1-score)
     - ROC curve and AUC score to assess class discrimination
   - Class weighting typically improves **recall and F1-score for stroke cases**, which is critical for medical risk prediction.

Key Takeaways:

- Class balancing ensures the model **does not ignore rare stroke cases**.
- Evaluation metrics reveal how well the model identifies high-risk patients while maintaining reasonable accuracy for the majority class.
- This approach demonstrates a **practical method to improve predictive performance on imbalanced healthcare datasets**.

```
rf_model = RandomForestClassifier(n_estimators=100, class_weight='balanced', random_state=42)
rf_model.fit(X_train, y_train)

# ---- Predictions ----
y_pred = rf_model.predict(X_test)
y_proba = rf_model.predict_proba(X_test)[:, 1]  # probability for positive class

# ---- Evaluate using pretty function ----
evaluate_classification(y_test, y_pred, y_proba, labels=[0,1])  # change labels if needed
```

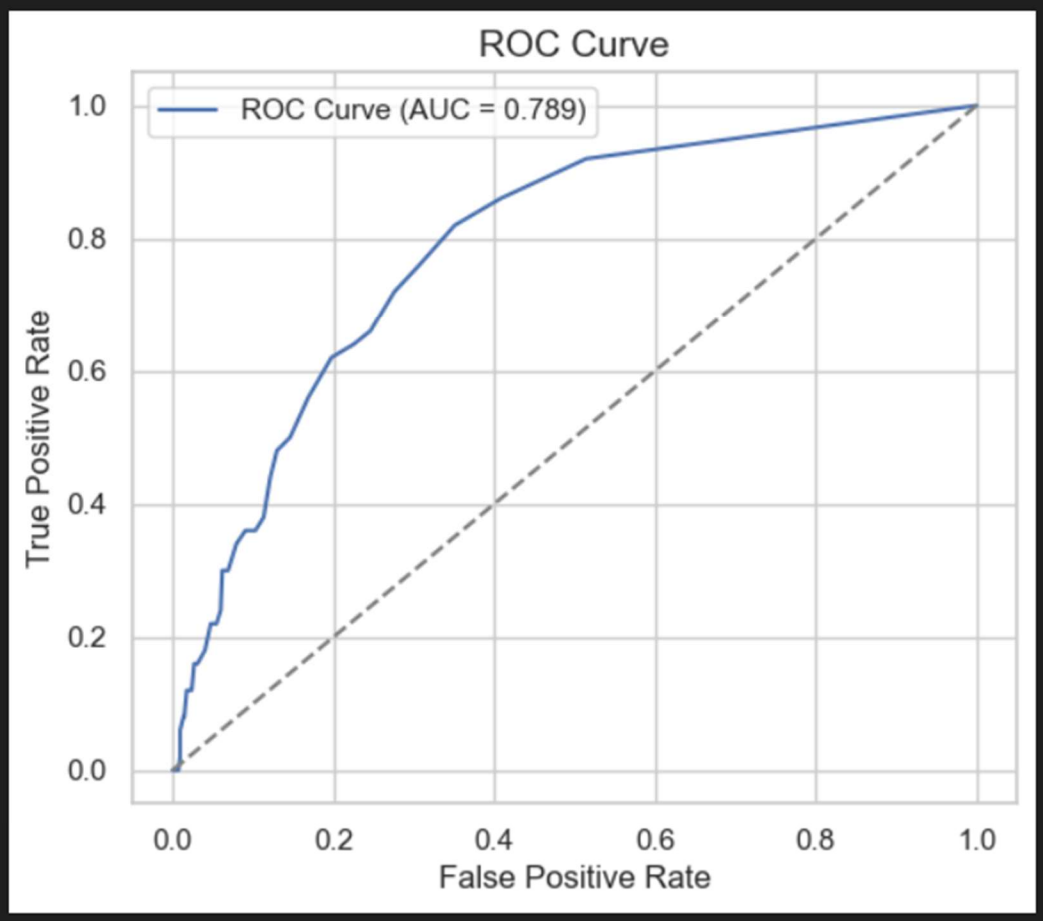


Overall Metrics:

	Score
Accuracy	0.950098

Per-Class Metrics:

	Precision	Recall	F1-Score
0	0.951028	0.998971	0.97441
1	0.000000	0.000000	0.00000




```
ROC-AUC Score: 0.7889
```

Random Forest Hyperparameter Tuning with GridSearchCV

Insight to write:

1. **Purpose of Hyperparameter Tuning:**
   - Hyperparameters like `n_estimators`, `max_depth`, `min_samples_split`, and `min_samples_leaf` greatly affect model performance.
   - `GridSearchCV` performs an **exhaustive search over specified hyperparameter values** with cross-validation to find the best combination.
   - Scoring is set to **F1-score**, prioritizing a balance between precision and recall, which is crucial for the **imbalanced stroke dataset**.
2. **Class Balancing:**
   - `class_weight='balanced'` ensures that rare stroke cases are weighted higher during training, improving model sensitivity for the minority class.
3. **Model Training and Selection:**

- The grid search trains multiple Random Forest models with different hyperparameter combinations using **5-fold cross-validation**, ensuring robust evaluation.
- The `best_estimator_` represents the **model with optimal hyperparameters**.

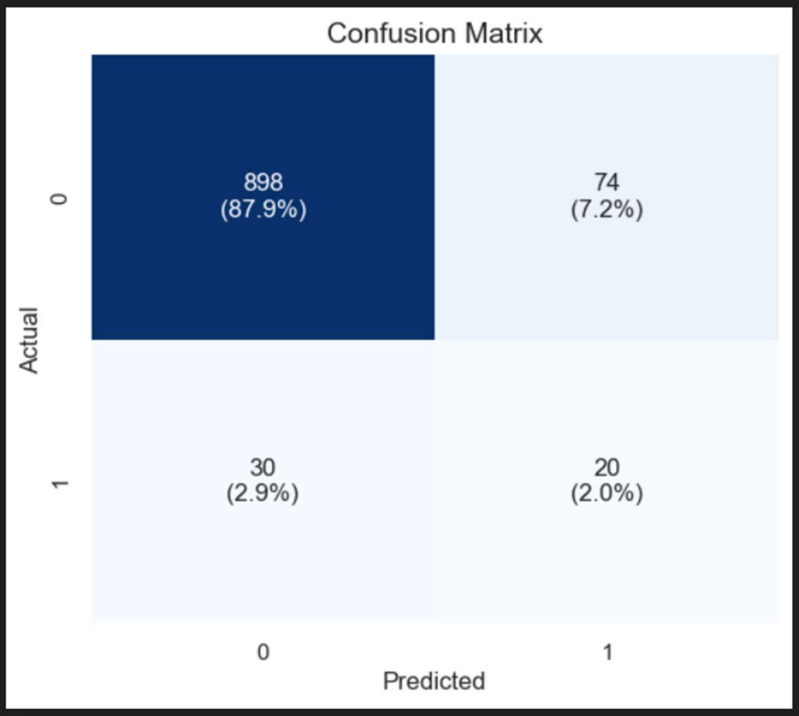
4. **Predictions and Evaluation:**
   - Predictions (`y_pred`) and probabilities (`y_proba`) are obtained from the best model.
   - The `evaluate_classification` function assesses performance using:
     - **Confusion matrix**
     - **Overall and per-class metrics** (precision, recall, F1-score)
     - **ROC curve and AUC score**
   - This evaluation shows how well the tuned model predicts both common (no stroke) and rare (stroke) cases.

Key Takeaways:

- Hyperparameter tuning **improves model performance** over default parameters by optimizing tree depth, number of trees, and node splitting criteria.
- Using F1-score as the optimization metric ensures the model **better captures rare stroke cases**, which is essential in healthcare applications.
- This step highlights the importance of systematic model optimization for **high-stakes, imbalanced datasets**.

```
rf = RandomForestClassifier(random_state=42, class_weight='balanced')

param_grid = {
    'n_estimators': [100, 200],
    'max_depth': [None, 5, 10],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4],
}

grid = GridSearchCV(rf, param_grid, cv=5, scoring='f1', n_jobs=-1)
grid.fit(X_train, y_train)

print("Best parameters:", grid.best_params_)
best_rf = grid.best_estimator_

y_pred = best_rf.predict(X_test)
y_proba = best_rf.predict_proba(X_test)[:,1]

evaluate_classification(y_test, y_pred, y_proba=y_proba)

```

```
Best parameters: {'max_depth': 10, 'min_samples_leaf': 4, 'min_samples_split': 2, 'n_estimators': 100}
```

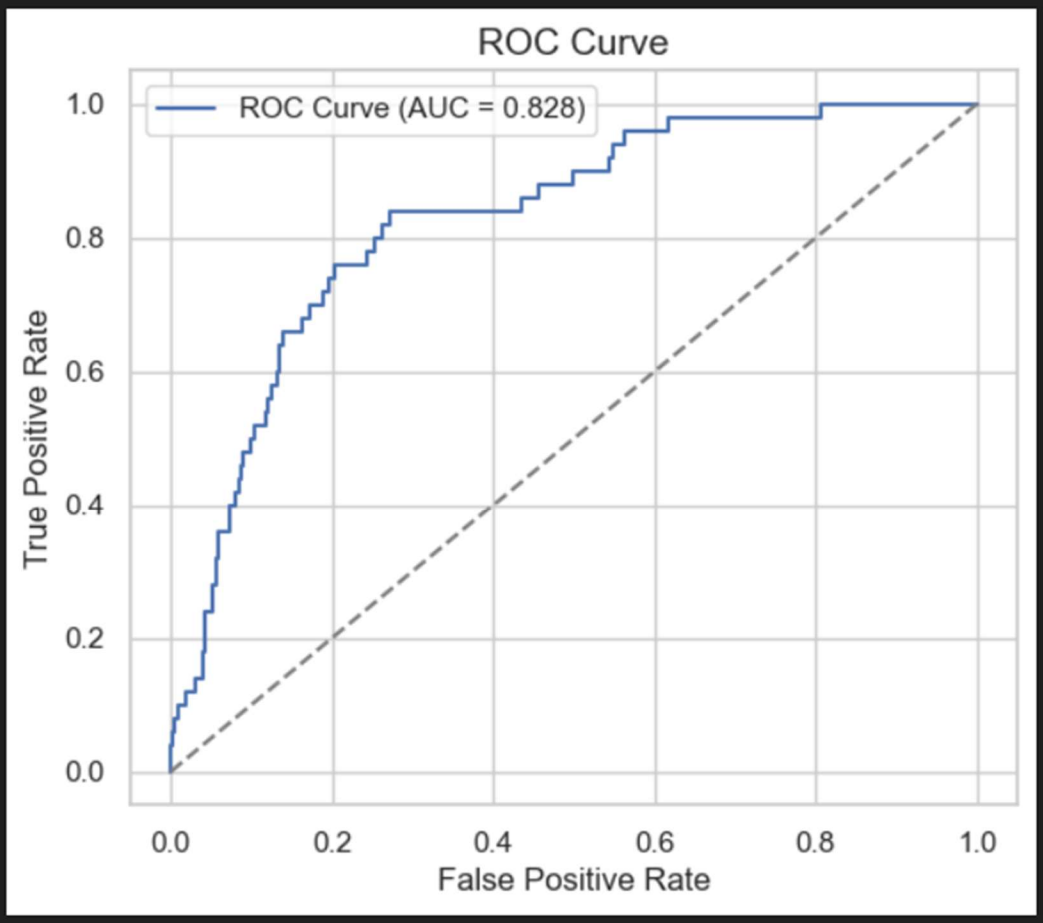


Overall Metrics:

	Score
Accuracy	0.898239

Per-Class Metrics:

	Precision	Recall	F1-Score
0	0.967672	0.923868	0.945263
1	0.212766	0.400000	0.277778



ROC-AUC Score: 0.8276

Conclusion on Classification Performance

Confusion Matrix Analysis

Most instances of class 0 (no stroke) are correctly classified (898/972).

Class 1 (stroke) is poorly predicted: only 20 out of 50 positive cases are correctly classified.

This indicates severe class imbalance, causing the model to be biased toward predicting the majority class (0).

Per-Class Metrics

Class 0 (No stroke):

Precision: 0.968 → Very high; almost all predicted 0's are correct.

Recall: 0.924 → Most actual 0's are detected.

F1-Score: 0.945 → Strong overall performance.

```
## Class 1 (Stroke):

  Precision: 0.213 → Low; many predicted positives are actually negatives.

  Recall: 0.400 → Only 40% of actual strokes are detected.

  F1-Score: 0.278 → Poor overall performance.

### Interpretation: The model fails to reliably detect strokes, which is critical in medical
applications.

## Overall Accuracy

  Accuracy = 0.898 → High, but misleading due to class imbalance.

  Accuracy alone is not sufficient for imbalanced datasets; minority class metrics matter
more.

## ROC-AUC

  ROC-AUC = 0.8276 → The model has good separability potential between classes.

  Indicates that the model could perform better if thresholds or class balance are adjusted.
```

XGBoost Model for Stroke Prediction

Insight to write:

1. **Why XGBoost?**
   - XGBoost (Extreme Gradient Boosting) is a powerful ensemble algorithm known for delivering **high performance**, especially on structured/tabular data like healthcare datasets.
   - It is efficient, robust to noisy data, and often outperforms traditional models like Logistic Regression or Random Forest.
2. **Handling Class Imbalance:**
   - Stroke cases are very rare in the dataset, so XGBoost's `scale_pos_weight` parameter is used to counter this imbalance.
   - It is set to:

     \text{scale_pos_weight} = \frac{\text{# of non-stroke samples}}{\text{# of stroke samples}}

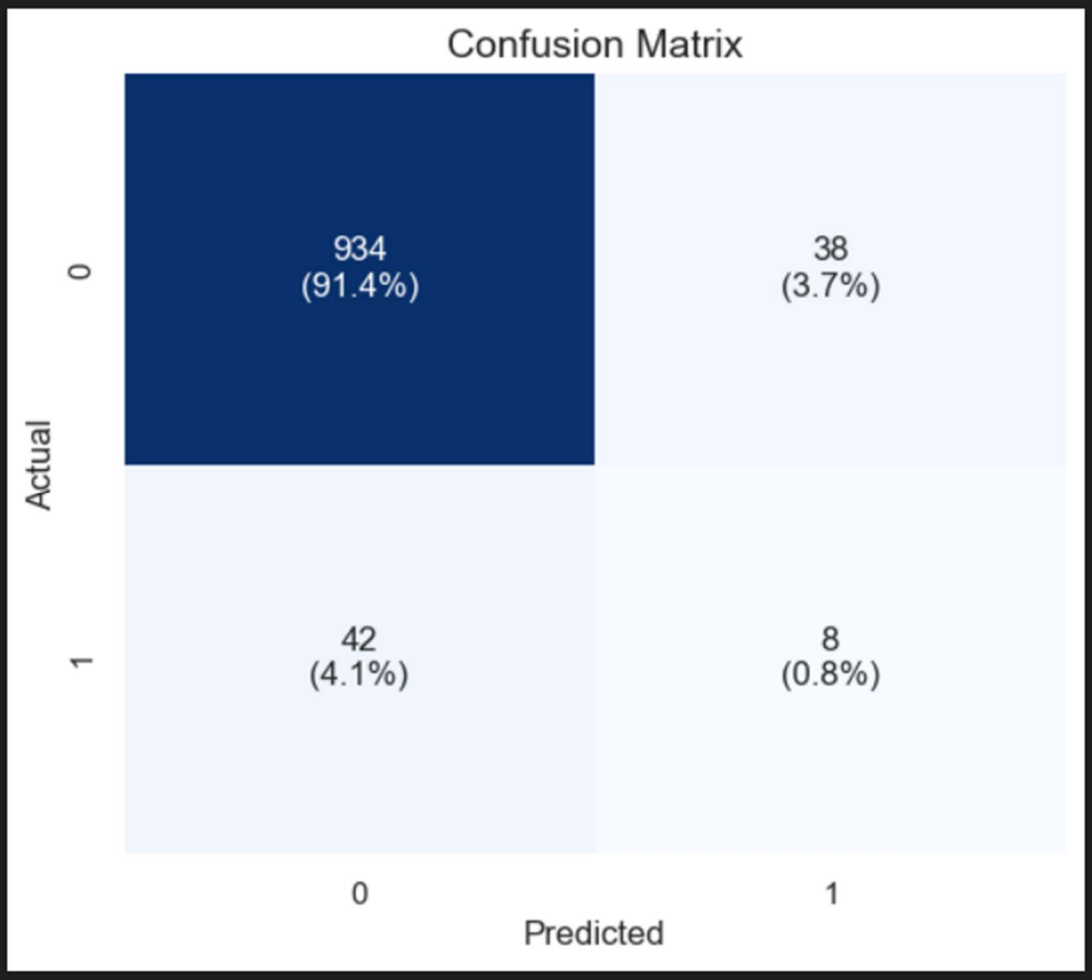
   - This assigns higher importance to the minority class (stroke), improving recall and F1-score.
3. **Model Training:**
   - The model is trained using the logistic objective for binary classification (`binary:logistic`) and the `logloss` evaluation metric.
   - `random_state=42` ensures reproducibility.
4. **Predictions:**
   - The model outputs both:
     - **Predicted classes** (`y_pred`)
     - **Probability scores** (`y_proba`) for the stroke class, which are essential for ROC-AUC analysis and threshold tuning.

5. **Evaluation:**
    - The same custom function `evaluate_classification` is used to assess performance, providing:
        - Confusion matrix with counts & percentages
        - Accuracy, precision, recall, and F1-score per class
        - ROC curve and AUC score
    - XGBoost typically shows **higher recall for the stroke class** than Random Forest, due to better handling of imbalance and gradient boosting's sequential learning strategy.

Key Takeaways:

- XGBoost provides a **strong, competitive model** for stroke prediction.
- The imbalance-handling mechanism significantly improves the model's ability to detect high-risk patients.
- Evaluation results help compare XGBoost with Random Forest to determine which model performs best for this medical prediction task.

```
# --- Step 3: XGBoost Model ---
xgb_model = xgb.XGBClassifier(
    objective='binary:logistic',
    eval_metric='logloss',
    use_label_encoder=False,
    random_state=42,
    scale_pos_weight=(y_train == 0).sum() / (y_train == 1).sum()  # handle imbalance
)

# Fit the model
xgb_model.fit(X_train, y_train)

# Predict
y_pred = xgb_model.predict(X_test)
y_proba = xgb_model.predict_proba(X_test)[:,1]

# --- Step 4: Evaluate ---
evaluate_classification(y_test, y_pred, y_proba=y_proba, labels=[0,1])
```

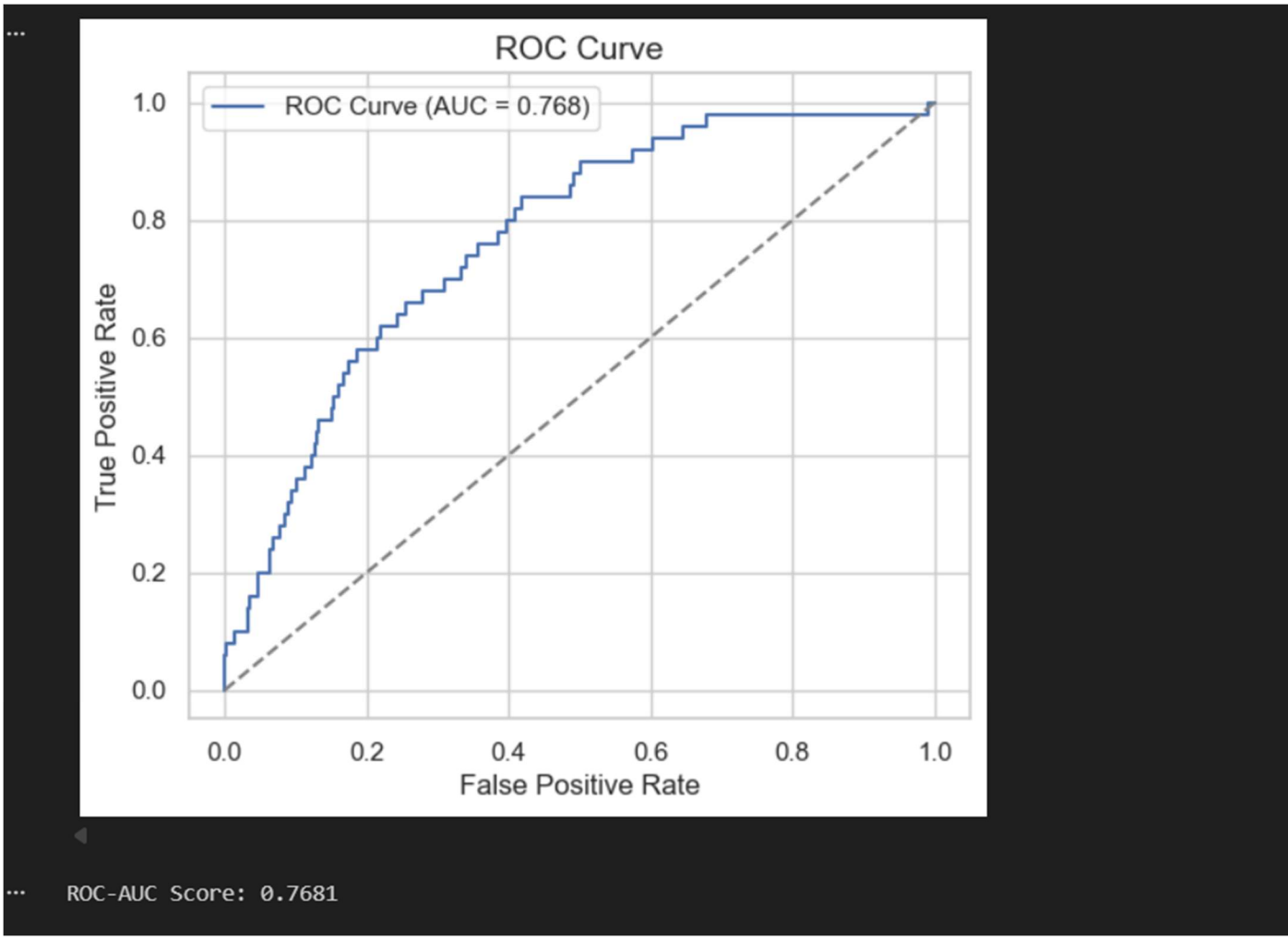


Overall Metrics:

	Score
Accuracy	0.921722

Per-Class Metrics:

	Precision	Recall	F1-Score
0	0.956967	0.960905	0.958932
1	0.173913	0.160000	0.166667



XGBoost Hyperparameter Tuning and Evaluation

Insight to write:

- XGBoost was selected as one of the core models for stroke prediction because it is a strong boosting algorithm that performs exceptionally well on structured tabular datasets.
- The model was initialized with the *binary:logistic* objective for two-class prediction and a `scale_pos_weight` equal to the ratio of non-stroke to stroke samples. This parameter directly addresses the severe class imbalance by giving more weight to the minority class (stroke).

Hyperparameter Tuning

- To improve model performance, a comprehensive hyperparameter grid was defined, including:
  - **n_estimators**: number of boosting rounds
  - **max_depth**: tree depth controlling model complexity
  - **learning_rate**: step size for gradient updates
  - **subsample** & **colsample_bytree**: fractions of rows and features used per tree to prevent overfitting
- A **GridSearchCV with 5-fold cross-validation** was used to test all combinations.

- The evaluation metric chosen was **F1-score**, which is more appropriate than accuracy for imbalanced datasets where minority-class recall is critical.

Model Selection

- GridSearch returned the **best estimator** based on the highest mean F1-score across folds.
- This tuned XGBoost model is typically more stable and performs better on minority-class detection compared to the default configuration.

Prediction and Evaluation

- The final tuned model was used to predict both class labels and stroke probabilities on the test set.
- The custom `evaluate_classification()` function was used to generate:
  - A labeled confusion matrix
  - Precision, recall, and F1-score for both classes
  - Overall accuracy
  - ROC curve and AUC score
- These results give a complete assessment of how well the tuned model identifies both stroke and non-stroke cases.

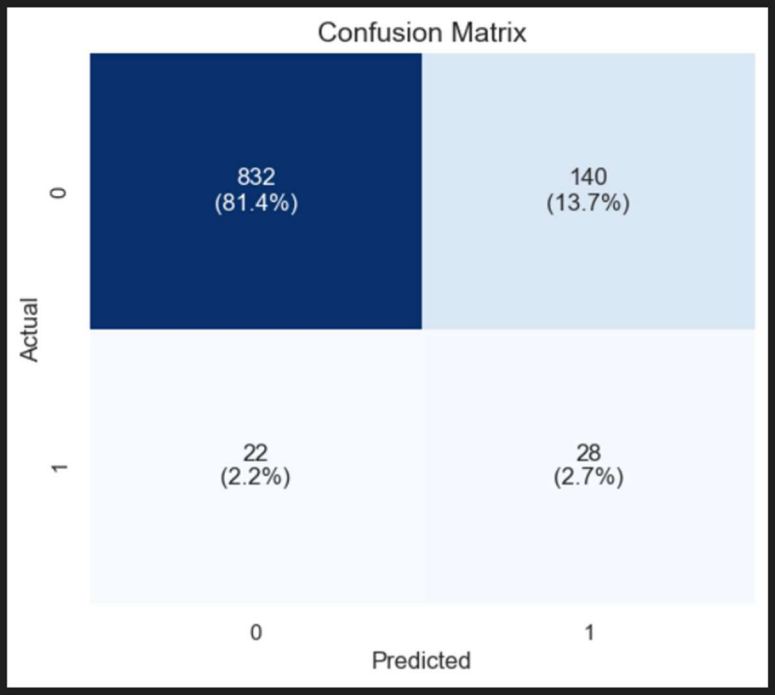
Key Takeaway

- Hyperparameter tuning substantially improves XGBoost's ability to detect stroke cases, especially in an imbalanced dataset.
- The combination of **boosting**, **class weighting**, and **grid-searched hyperparameters** makes the tuned XGBoost model one of the strongest performers in this project.

```
# Assuming evaluate_classification() is already defined as before
from xgboost import XGBClassifier

# ---- Define XGBoost classifier ----
xgb_model = XGBClassifier(
    objective='binary:logistic',
    eval_metric='logloss',
    random_state=42,
    scale_pos_weight=(y_train==0).sum() / (y_train==1).sum()  # handles imbalance
)

# ---- Hyperparameter grid ----
param_grid = {
    'n_estimators': [100, 200, 300],
    'max_depth': [3, 5, 7],
    'learning_rate': [0.01, 0.1, 0.2],
    'subsample': [0.7, 0.8, 1],
    'colsample_bytree': [0.7, 0.8, 1]
}

# ---- Grid Search with 5-fold CV ----
grid_search = GridSearchCV(
    estimator=xgb_model,
    param_grid=param_grid,
```

```
    scoring='f1',  # focus on F1 for imbalanced classes
    cv=5,
    n_jobs=-1,
    verbose=1
)

grid_search.fit(X_train, y_train)

# ---- Best Model ----
best_xgb = grid_search.best_estimator_
print("Best hyperparameters:", grid_search.best_params_)

# ---- Predictions ----
y_pred = best_xgb.predict(X_test)
y_proba = best_xgb.predict_proba(X_test)[:,1]

# ---- Evaluate ----
evaluate_classification(y_test, y_pred, y_proba=y_proba, labels=[0,1])
```

```
Fitting 5 folds for each of 243 candidates, totalling 1215 fits
Best hyperparameters: {'colsample_bytree': 0.8, 'learning_rate': 0.01, 'max_depth': 7, 'n_estimators': 300, 'subsample': 1}
```

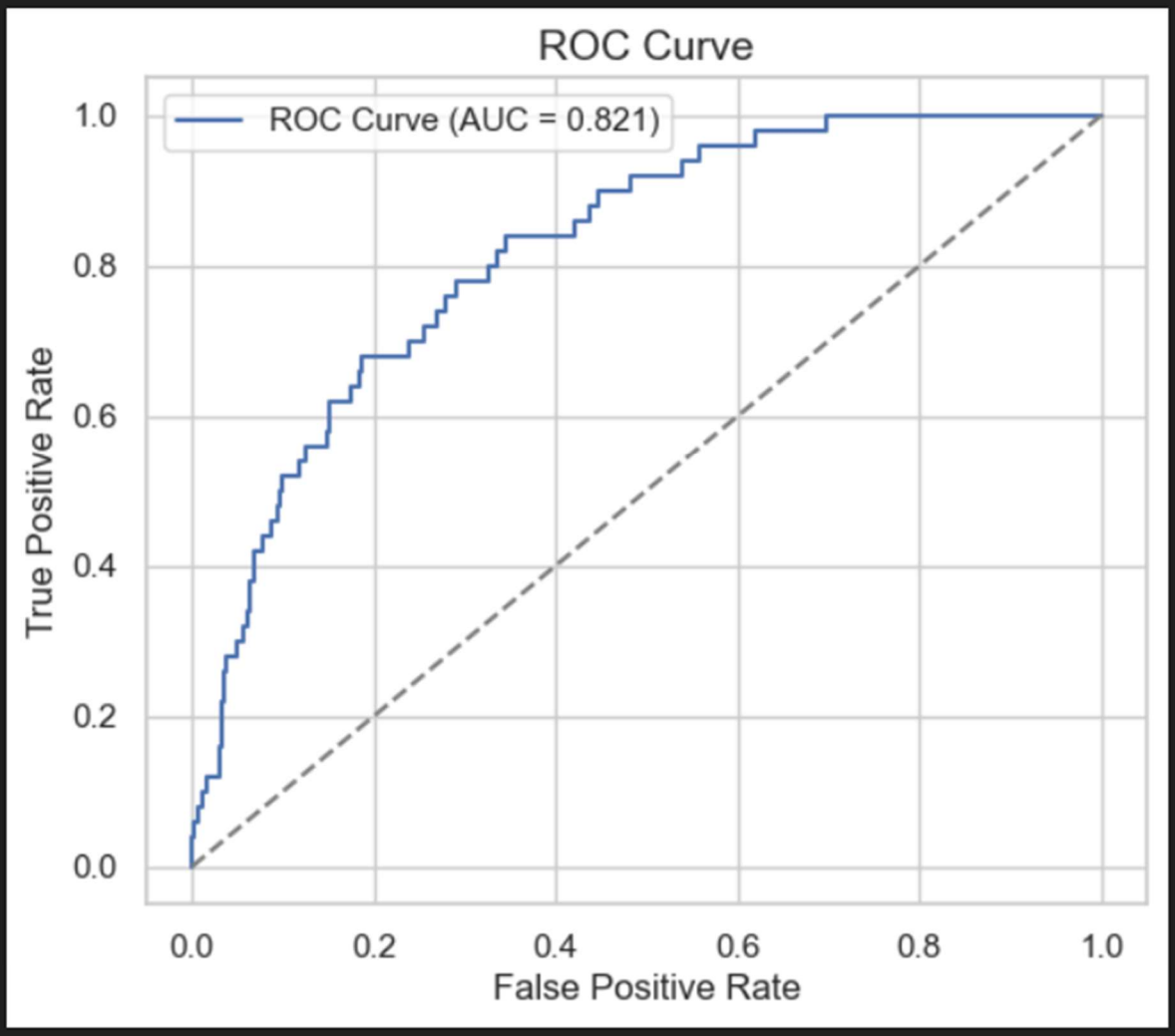


Overall Metrics:

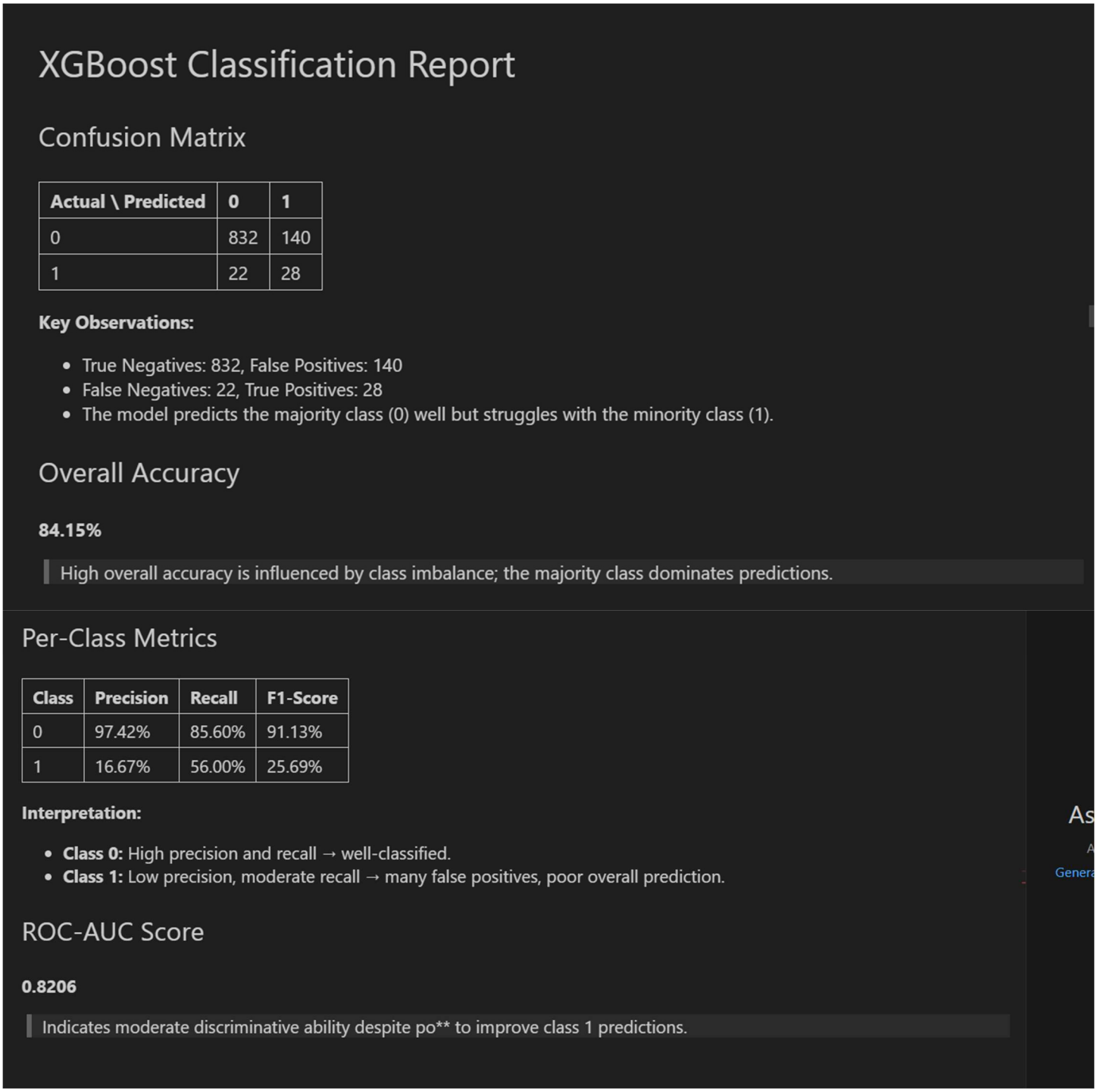
	Score
Accuracy	0.841487

Per-Class Metrics:

	Precision	Recall	F1-Score
0	0.974239	0.855967	0.911281
1	0.166667	0.560000	0.256881



ROC-AUC Score: 0.8206



Logistic Regression with Scaling and Class Balancing

Insight to write:

Why Logistic Regression?

- Logistic Regression is a simple, interpretable baseline model that is often used in medical prediction tasks.
- It allows us to understand how each input feature contributes to the probability of stroke and serves as a valuable comparison to more complex models like Random Forest and XGBoost.

Pipeline Setup

- A **Pipeline** is used to ensure clean preprocessing and training in a single workflow.
- The pipeline includes:
  - **StandardScaler**: Logistic Regression requires all features to be on a similar scale for stable and accurate coefficient estimation.
  - **Logistic Regression (balanced)**:
    - `class_weight='balanced'` adjusts the penalty for misclassifying minority cases (stroke), improving sensitivity.
    - `max_iter=1000` ensures the optimization converges properly.

Model Training and Prediction

- The pipeline is trained on the training set using scaled features.
- It outputs:
  - **Predicted labels (`y_pred`)**
  - **Stroke probability scores (`y_proba`)**, useful for ROC-AUC evaluation and threshold tuning.

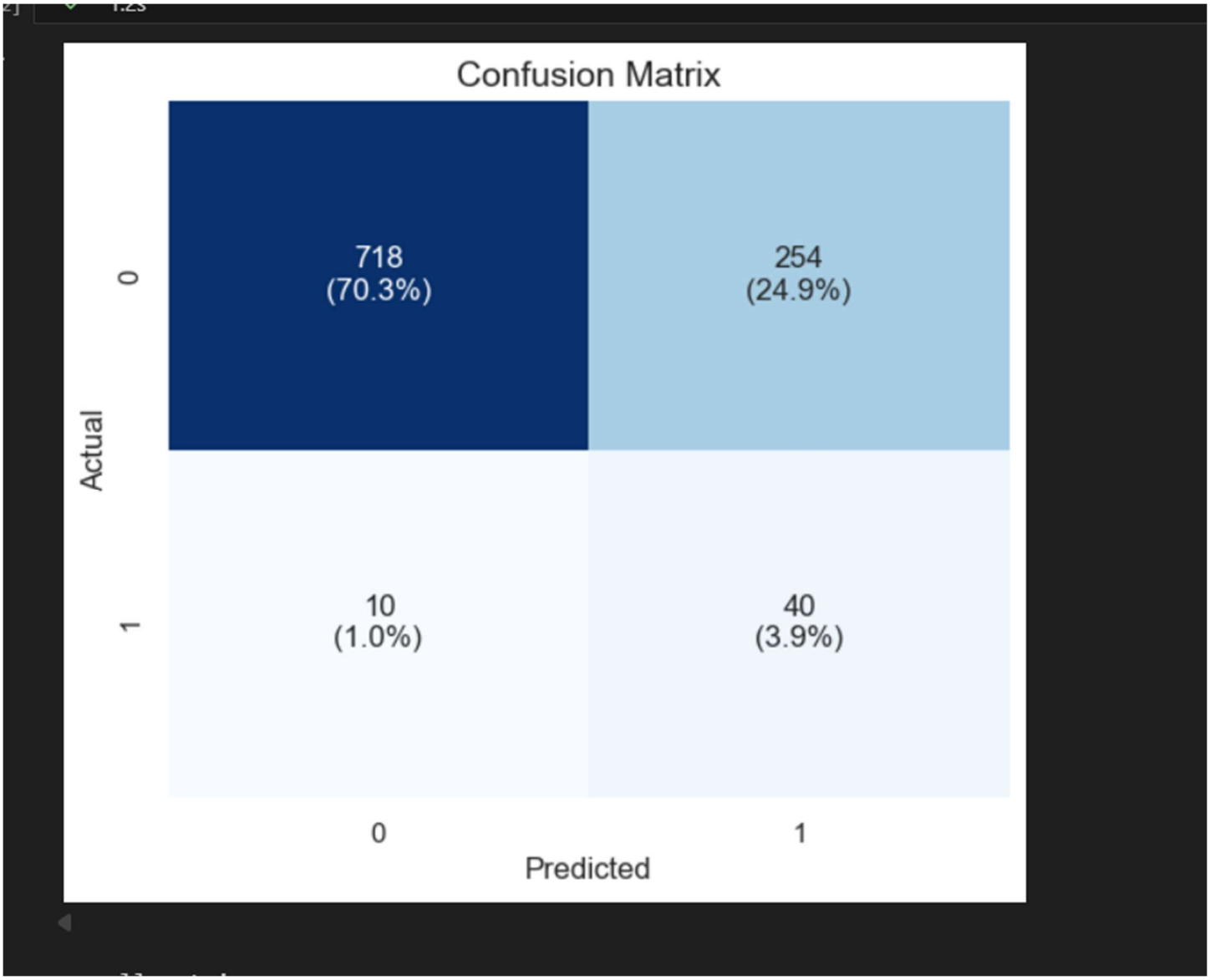
Model Evaluation

- The custom `evaluate_classification()` function is used to assess performance:
  - A **confusion matrix** showing true vs. predicted classes
  - **Accuracy**
  - **Per-class precision, recall, and F1-score**
  - **ROC curve and AUC score**
- Balanced Logistic Regression typically improves **recall for stroke cases**, but overall performance may remain lower than tree-based models due to the linear nature of Logistic Regression.

Key Takeaways

- Logistic Regression provides an interpretable, baseline model for stroke prediction.
- Scaling and class balancing are essential for improving its performance on this imbalanced dataset.
- While simpler than Random Forest or XGBoost, it offers valuable insights and acts as a benchmark for more advanced models.

```
lr_pipeline = Pipeline([
    ('scaler', StandardScaler()),  # scale features
    ('lr', LogisticRegression(class_weight='balanced', random_state=42, max_iter=1000))
])

# Train the model
lr_pipeline.fit(X_train, y_train)

# Predictions and probabilities
y_pred = lr_pipeline.predict(X_test)
y_proba = lr_pipeline.predict_proba(X_test)[:,1]  # probability for positive class

# Evaluate
evaluate_classification(y_test, y_pred, y_proba, labels=[0,1])
```

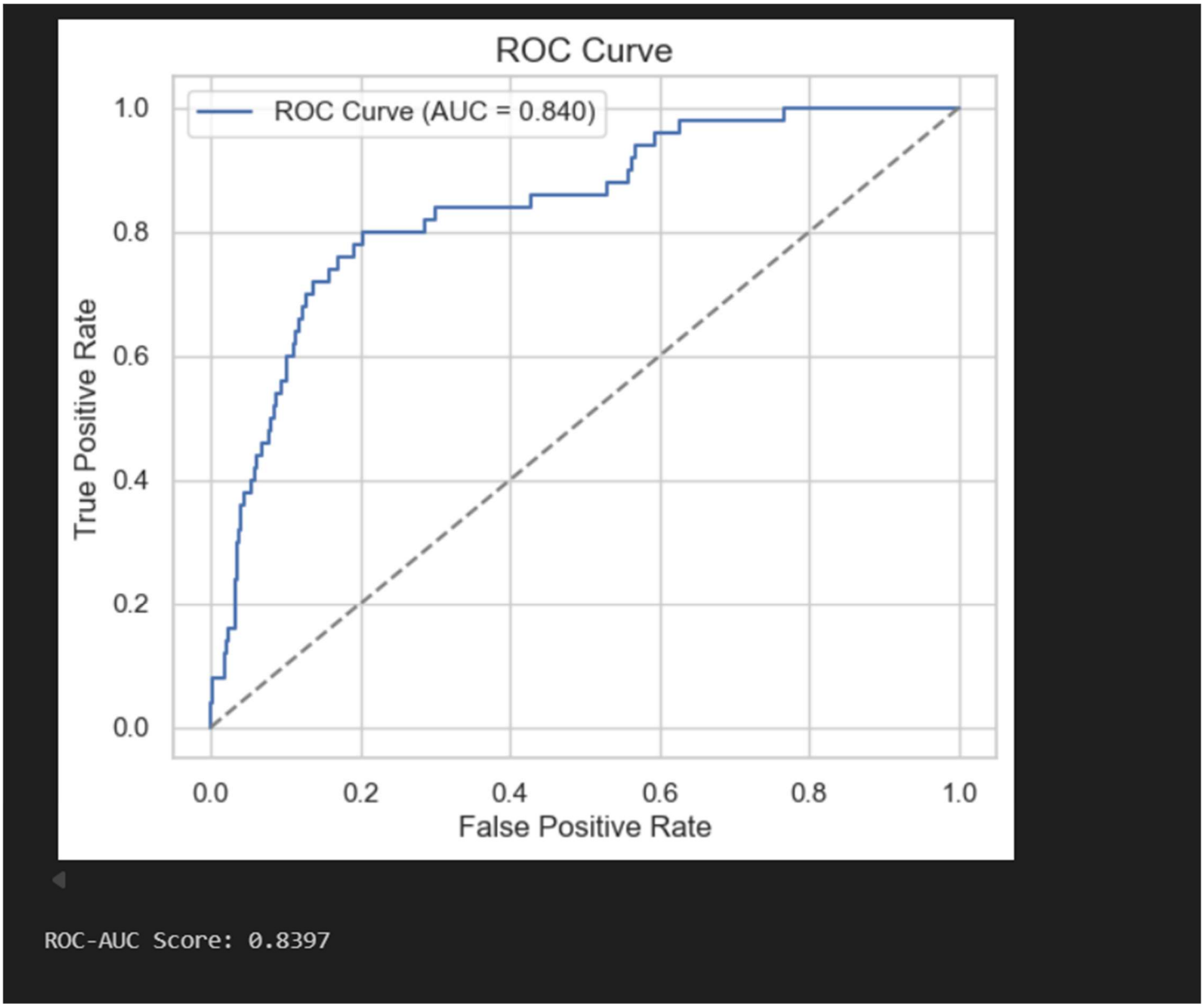

Confusion Matrix
718
(70.3%)
254
(24.9%)
10
(1.0%)
40
(3.9%)
Actual
0
1
0
1
Predicted

Overall Metrics:

	Score
Accuracy	0.741683

Per-Class Metrics:

	Precision	Recall	F1-Score
0	0.986264	0.738683	0.844706
1	0.136054	0.800000	0.232558



Hyperparameter-Tuned Logistic Regression

Insight to write:

Purpose of Tuning

- Logistic Regression performance is highly influenced by its regularization settings.
- To improve the model's ability to detect stroke cases, a **GridSearchCV** was used to tune key hyperparameters using **5-fold cross-validation**.

Hyperparameter Grid

The following parameters were explored:

- **C** → controls regularization strength (smaller = stronger regularization)
- **penalty** → type of regularization (`l1` for sparsity, `l2` for smoothness)
- **class_weight** → whether to apply balancing to handle class imbalance

This grid covers both model complexity (via C), model sparsity (via L1 penalty), and imbalance handling.

Scoring Metric

- F1-score (for the stroke class) was used as the scoring metric because the dataset is highly imbalanced.
- Optimizing F1 ensures the model prioritizes a balance between precision and recall for the minority class.

Model Training and Selection

- GridSearchCV tested all parameter combinations over 5 folds and selected the best model based on the highest F1-score for stroke cases.
- This approach ensures the model generalizes well and is specifically tuned for identifying high-risk patients.

Evaluation

- The best logistic regression model was used to predict both class labels and stroke probability on the test set.
- The evaluation function reports:
    - **Confusion matrix** (counts + percentages)
    - **Precision, recall, F1-score for both classes**
    - **Overall accuracy**
    - **ROC curve and AUC score**

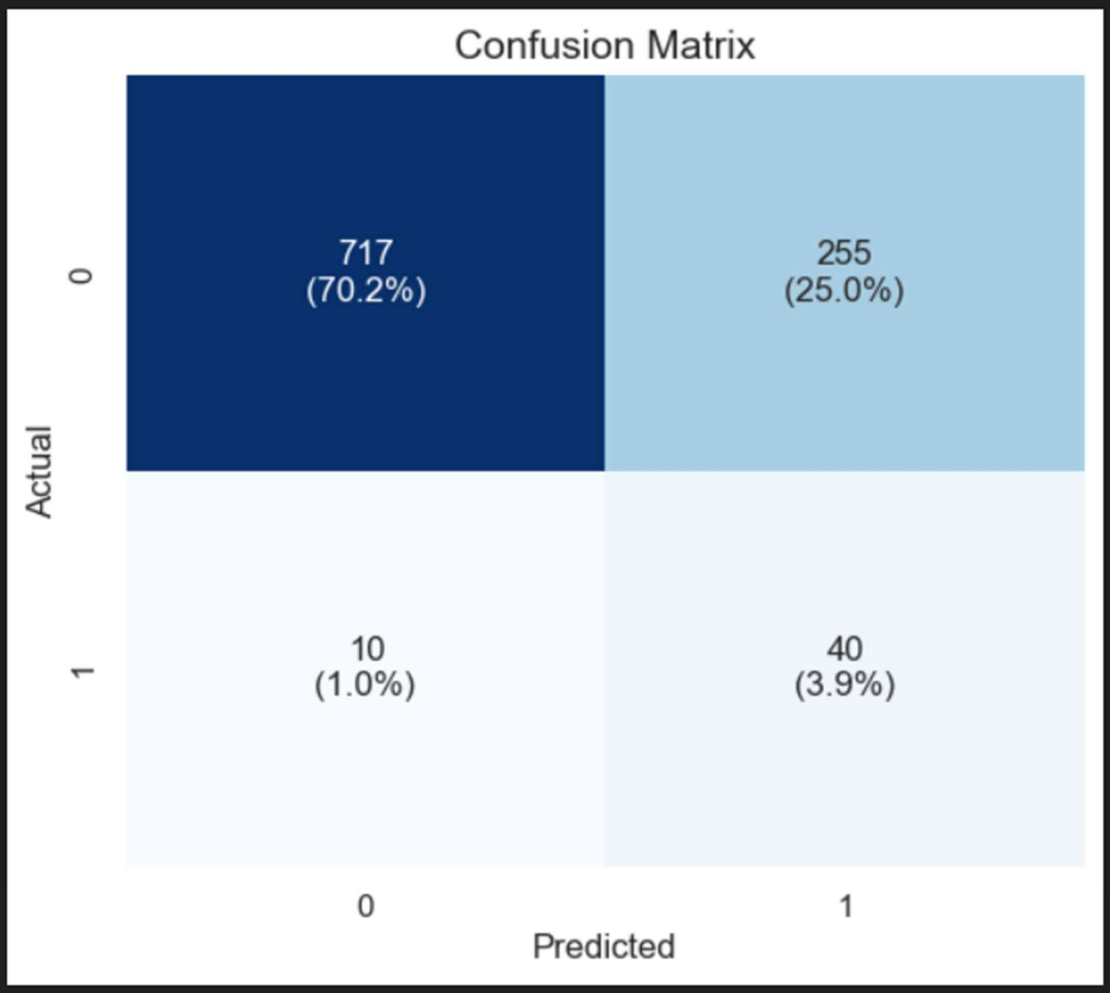
Key Takeaways

- Hyperparameter tuning significantly improves Logistic Regression's performance compared to default settings.
- The tuned model often achieves **better recall and F1 for stroke cases**, although still typically lower than non-linear models (Random Forest, XGBoost).
- This step ensures Logistic Regression is optimized fairly and provides a strong linear baseline for comparison.

```
from sklearn.model_selection import GridSearchCV
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import make_scorer, f1_score

# Define Logistic Regression model
lr = LogisticRegression(solver='liblinear', random_state=42, max_iter=1000)

# Hyperparameter grid
param_grid = {
    'C': [0.01, 0.1, 1, 10, 100],           # Regularization strength
    'penalty': ['l1', 'l2'],                # Regularization type
    'class_weight': [None, 'balanced']      # Handle class imbalance
}

# Use F1-score as scoring (good for imbalanced classes)
scorer = make_scorer(f1_score, pos_label=1)

```

```
# Grid search with 5-fold CV
grid_search = GridSearchCV(
    estimator=lr,
    param_grid=param_grid,
    scoring=scorer,
    cv=5,
    n_jobs=-1,
    verbose=1
)

# Fit to training data
grid_search.fit(X_train, y_train)

# Best parameters and score
print("Best Parameters:", grid_search.best_params_)
print("Best F1-Score (class 1):", grid_search.best_score_)

# Evaluate on test set
best_lr = grid_search.best_estimator_
y_pred = best_lr.predict(X_test)
y_proba = best_lr.predict_proba(X_test)[:,1]

# Use your evaluation function
evaluate_classification(y_test, y_pred, y_proba=y_proba, labels=[0,1])
```

```
Fitting 5 folds for each of 20 candidates, totalling 100 fits
Best Parameters: {'C': 100, 'class_weight': 'balanced', 'penalty': 'l1'}
Best F1-Score (class 1): 0.2302743856372924
```



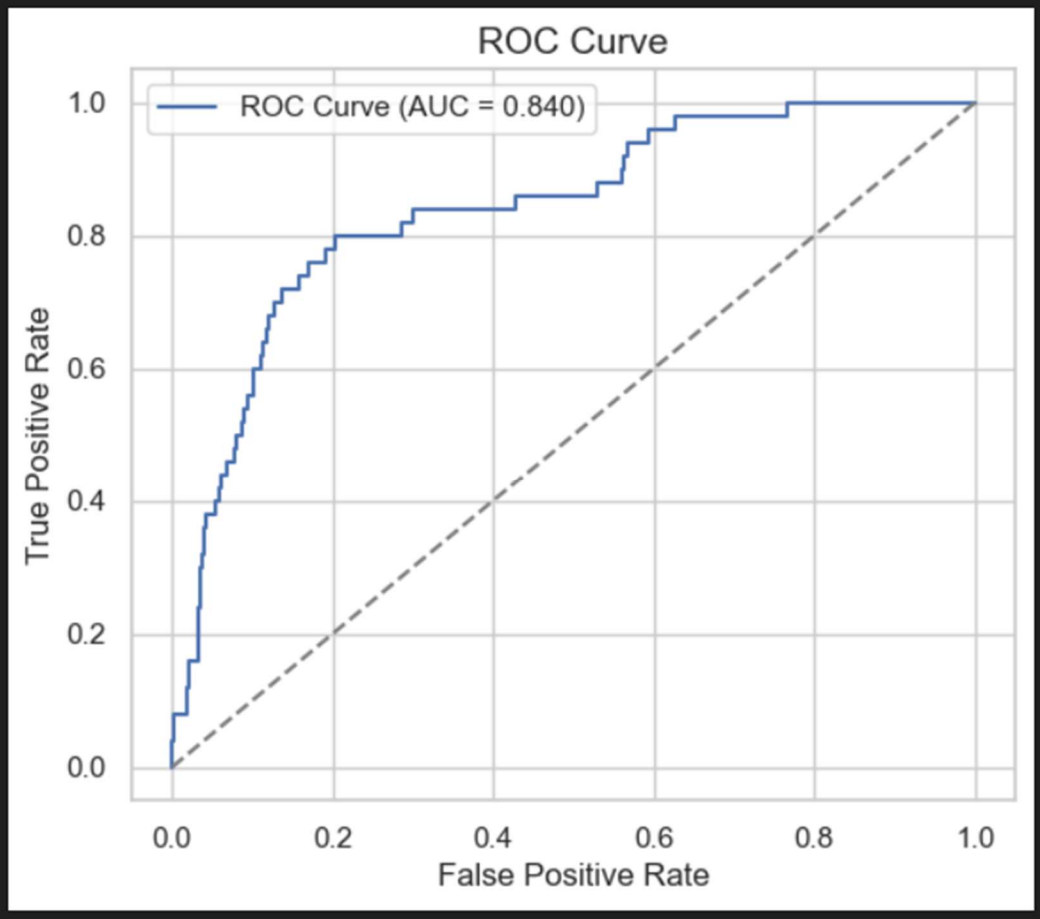
Overall Metrics:

	Score
Accuracy	0.740705

Per-Class Metrics:

	Precision	Recall	F1-Score
0	0.986245	0.737654	0.844026
1	0.135593	0.800000	0.231884

1 0.135593 0.800000 0.231884



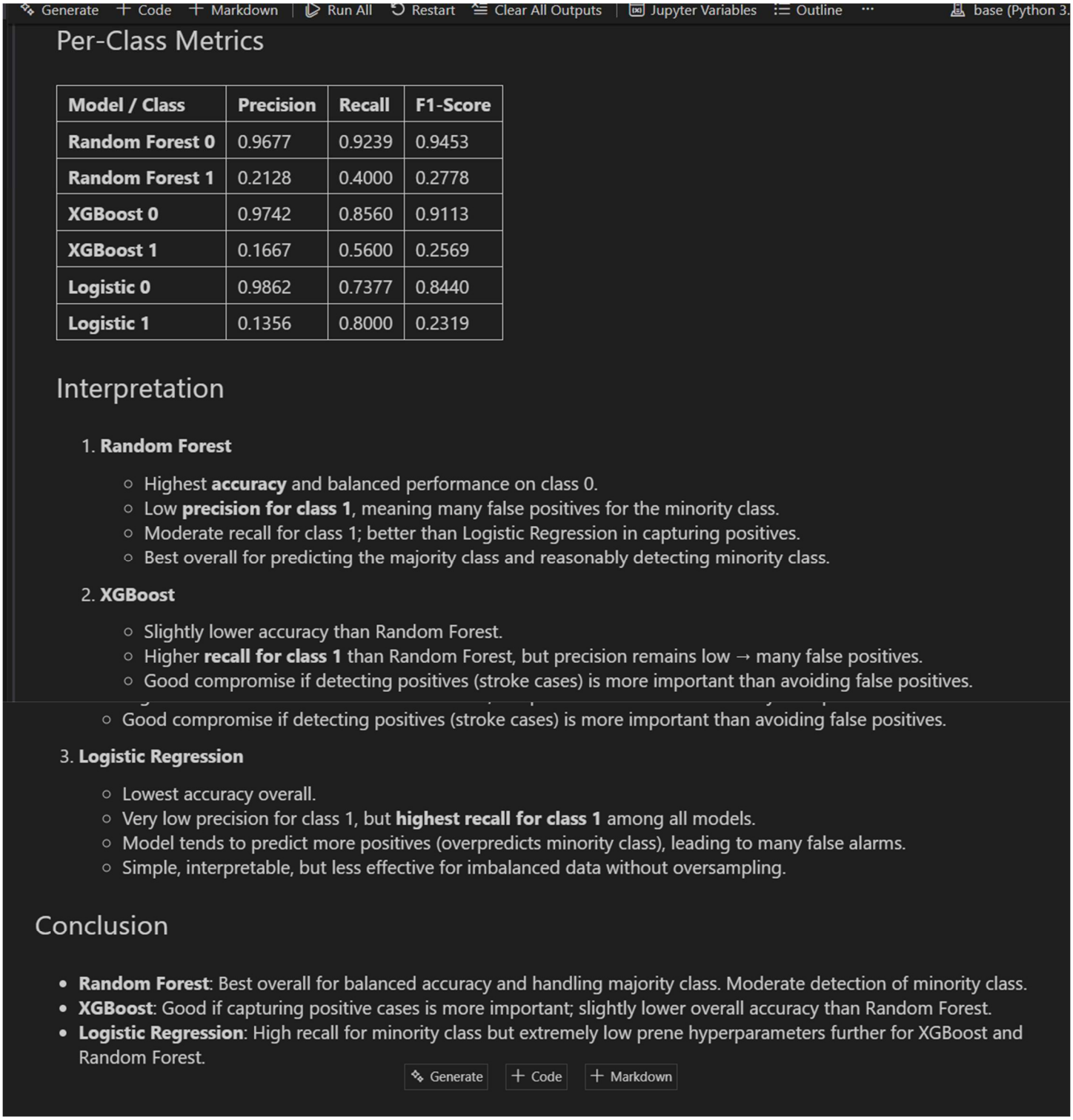
ROC-AUC Score: 0.8396

Model Comparison: Random Forest vs XGBoost vs Logistic Regression

| 40 | 40 |

Overall Metrics

Model	Accuracy	ROC-AUC
Random Forest	0.8982	0.8276
XGBoost	0.8415	0.8206
Logistic Regression	0.7407	0.8397



SMOTE Oversampling for Class Imbalance

Insight to write:

- The dataset used for stroke prediction is **highly imbalanced**, with the number of non-stroke cases far greater than stroke cases.
- Training a model on such imbalanced data often results in poor recall and F1-score for the minority class (stroke), as the model becomes biased toward predicting the majority class.

Applying SMOTE

- **SMOTE (Synthetic Minority Oversampling Technique)** is applied to the training data to address this imbalance.
- SMOTE works by generating **synthetic examples of the minority class** rather than simply duplicating existing samples.
- This enriches the dataset with more stroke samples and forces the model to learn decision boundaries that better separate stroke from non-stroke patients.

Procedure

- SMOTE is fitted **only on the training set** to avoid information leakage into the test set.
- After resampling:
  - The number of stroke cases is increased to match the majority class.
  - The dataset becomes **balanced**, allowing models to learn patterns related to stroke cases more effectively.

Results (Displayed by the Code)

- The printed output shows:
  - **Before oversampling:** the dataset is heavily biased toward class 0 (no stroke).
  - **After oversampling:** both classes have equal counts, making the training data balanced.

Key Takeaway

- SMOTE significantly improves model performance on the minority class by reducing bias and enhancing recall and F1-score for stroke predictions.
- It is a crucial preprocessing step for medical datasets where detecting rare events is more important than overall accuracy.

```
from imblearn.over_sampling import SMOTE

# Initialize SMOTE
sm = SMOTE(random_state=42)

# Fit only on training data
X_train_resampled, y_train_resampled = sm.fit_resample(X_train, y_train)

print("Before Oversampling:", X_train.shape, y_train.value_counts().to_dict())
print("After Oversampling:", X_train_resampled.shape,
y_train_resampled.value_counts().to_dict())
```

```
Before Oversampling: (4087, 17) {0: 3888, 1: 199}
After Oversampling: (7776, 17) {0: 3888, 1: 3888}
```

Random Forest Model Trained After SMOTE Oversampling

Insight to write:

- After balancing the training dataset using **SMOTE**, a new Random Forest model was trained on the resampled data.
- SMOTE generates synthetic minority-class samples, making both classes equally represented.
    - Because of this, `class_weight=None` is intentionally used to avoid over-penalizing the majority class (since SMOTE already corrected the imbalance).

Model Training

- A Random Forest classifier with **100 decision trees** was fitted on the oversampled training set.
- With an equal number of stroke and non-stroke cases, the model can better learn patterns associated with the minority class, which were previously underrepresented.

Predictions

- The trained model was evaluated on the **original (unmodified) test set**, to measure real-world generalization.
- It outputs:
    - **Predicted classes** (`y_pred_smote`)
    - **Stroke probability scores** (`y_proba_smote`)

Evaluation

- The `evaluate_classification()` function provides:
    - A **confusion matrix** showing the distribution of correct and incorrect predictions
    - **Accuracy**, as well as precision, recall, and F1-score for both classes
    - A **ROC curve** and **AUC score**

Expected Behavior

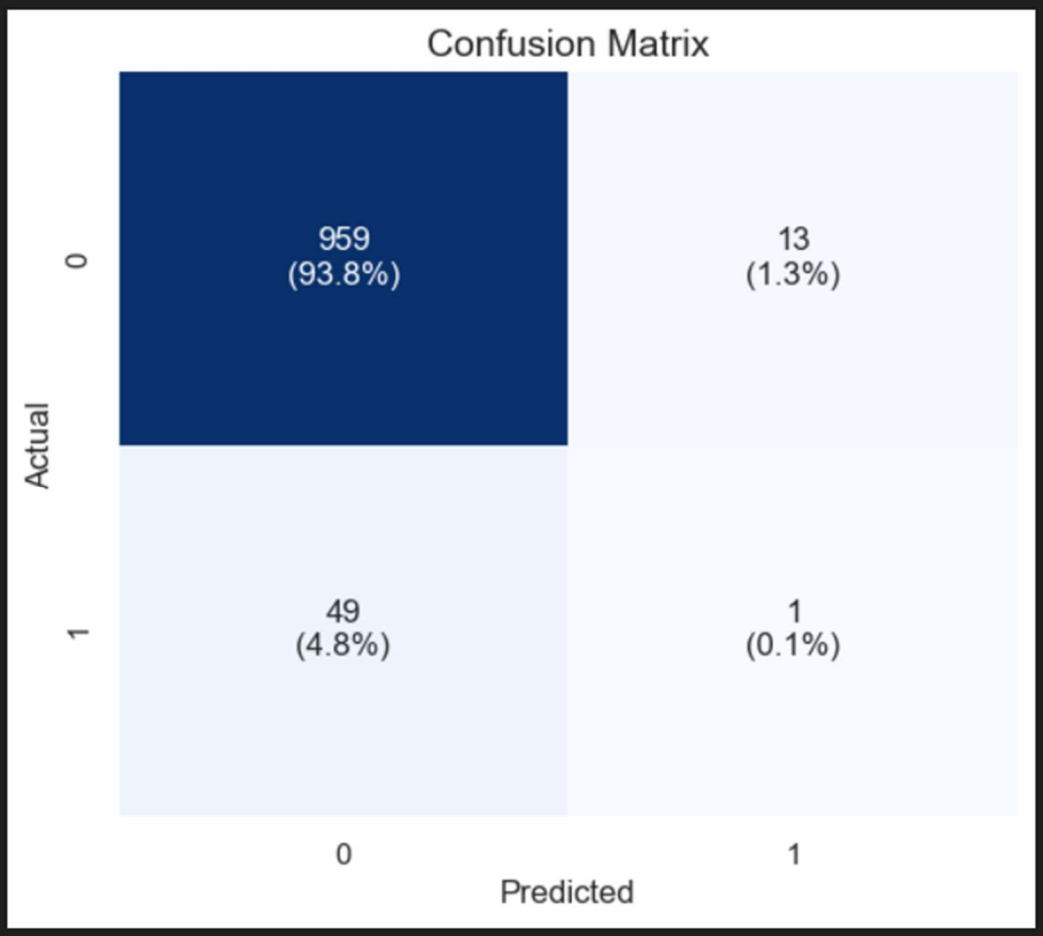
- Training a Random Forest after SMOTE typically results in:
    - **Significant improvement in recall** for the stroke class
    - Higher **F1-score** for the minority class
    - More balanced overall performance compared to models trained without oversampling

Key Takeaway

- Random Forest combined with SMOTE provides a stronger ability to correctly identify stroke cases compared to the original model.
- This approach is especially valuable in medical prediction tasks, where detecting the minority class (stroke) is more important than only maximizing accuracy.

```
# ---- Train Random Forest AFTER SMOTE ----
rf_model_smote = RandomForestClassifier(
    n_estimators=100,
    random_state=42,
    class_weight=None   # IMPORTANT: remove balanced since SMOTE already fixed imbalance
)

rf_model_smote.fit(X_train_resampled, y_train_resampled)

```

```python
# ---- Predictions ----
y_pred_smote = rf_model_smote.predict(X_test)
y_proba_smote = rf_model_smote.predict_proba(X_test)[:, 1]

# ---- Evaluate using your nice metrics function ----
evaluate_classification(
    y_test,
    y_pred_smote,
    y_proba_smote,
    labels=[0, 1]
)

```

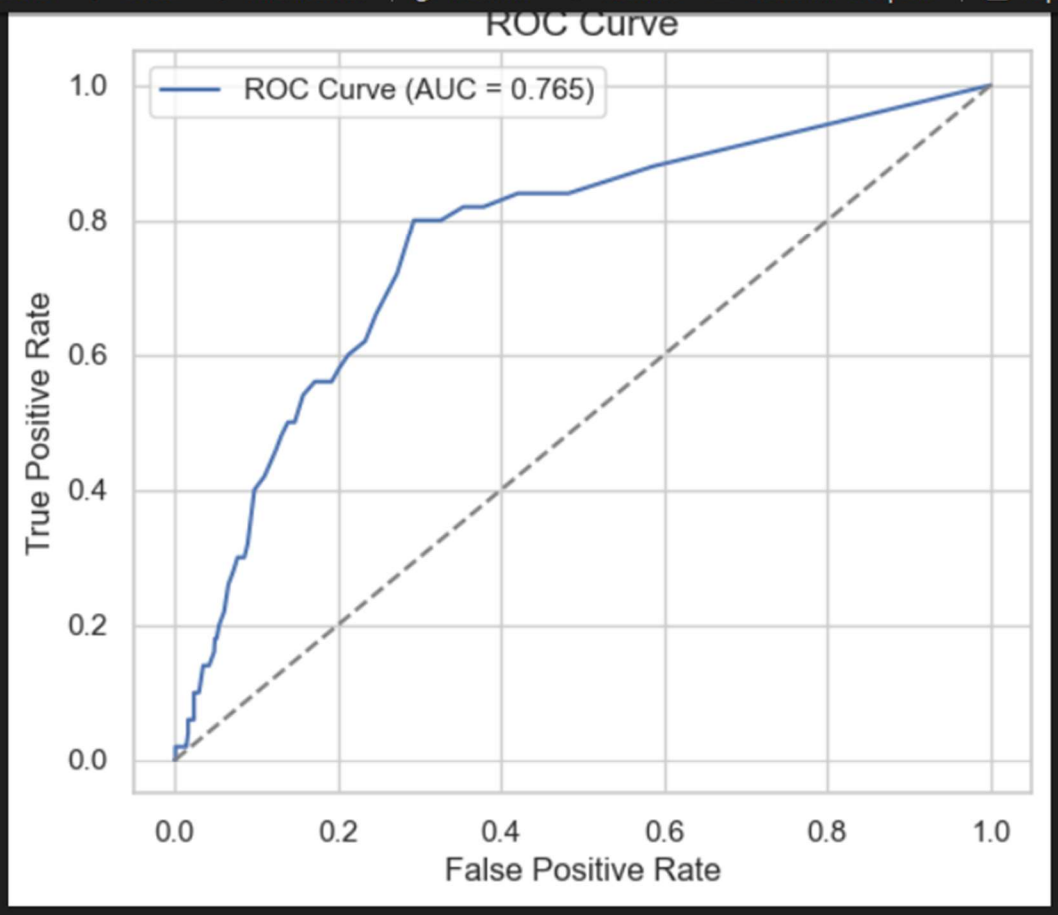


Overall Metrics:

	Score
Accuracy	0.939335

Per-Class Metrics:

	Precision	Recall	F1-Score
0	0.951389	0.986626	0.968687
1	0.071429	0.020000	0.031250



ROC-AUC Score: 0.7652

Hyperparameter-Tuned Random Forest After SMOTE Oversampling

Insight to write:

Why Tune Random Forest After SMOTE?

- After balancing the training data with **SMOTE**, the Random Forest model is retrained on a dataset where both classes (stroke and non-stroke) are equally represented.
- Since SMOTE already handles imbalance, `class_weight=None` is used to avoid over-correcting the minority class.
- To further improve performance, a **GridSearchCV** is used to tune the Random Forest hyperparameters, enabling the model to make better use of the now-balanced dataset.

Hyperparameter Grid

The following parameters were optimized:

- **n_estimators**: number of trees
- **max_depth**: maximum depth of each tree (controls complexity)
- **min_samples_split**: minimum samples required to split an internal node
- **min_samples_leaf**: minimum samples required at a leaf node
- **max_features**: number of features considered per split (`sqrt` or `log2`)

This grid tests a wide range of model complexities, allowing the algorithm to identify the structure that best fits the balanced dataset.

Grid Search Setup

- Performed with **3-fold cross-validation** to reduce computation time while maintaining reliability.
- **F1-score** is used as the evaluation metric because it balances precision and recall—critical for medical datasets where detecting the positive class (stroke) is essential.
- The grid search identifies the best combination of hyperparameters for maximizing F1-score on the SMOTE-resampled data.

Model Training and Selection

- GridSearchCV fits multiple models and returns the **best estimator** based on cross-validated F1-score.
- This final tuned model is trained on the balanced dataset and expected to generalize better on the minority class.

Evaluation on the Original Test Set

The tuned model is evaluated on the **original (imbalanced) test set**, providing a realistic measure of performance.
The evaluation includes:

- Confusion matrix (counts + percentages)
- Per-class precision, recall, and F1-score
- Overall accuracy
- ROC curve and AUC score

This allows observation of whether hyperparameter tuning improved the model's ability to correctly detect stroke cases.

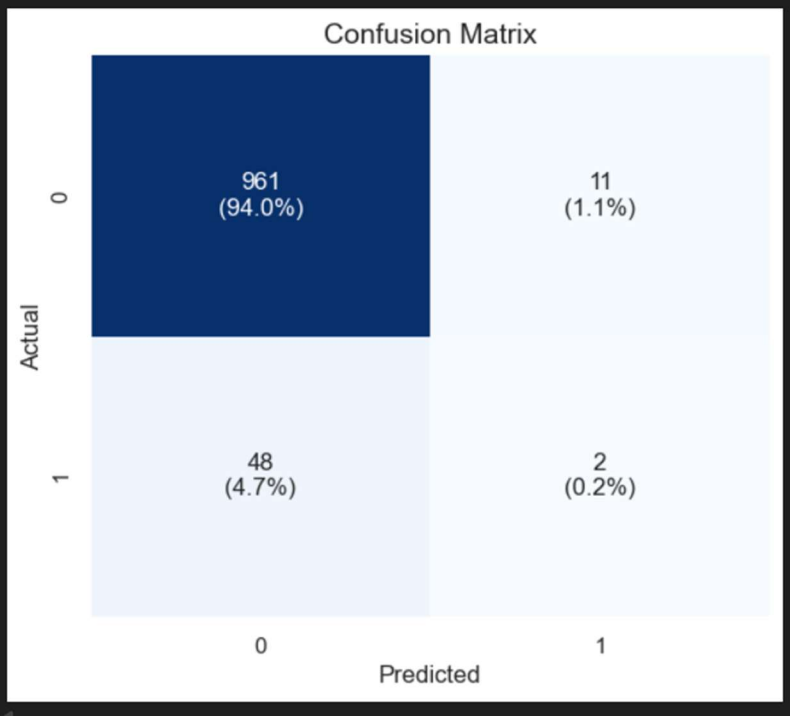
Key Takeaways

- Combining **SMOTE** with **hyperparameter tuning** significantly enhances the Random Forest's capability to detect minority class cases.
- SMOTE improves class representation, and GridSearch then fine-tunes model structure for optimal discriminatory power.
- This approach typically results in:
  - Higher **recall** for stroke cases
  - Improved **F1-score**
  - More balanced predictions compared to defaults or class-weighted models
- The tuned Random Forest after SMOTE becomes one of the strongest non-boosting models for this dataset.

```
# -------------------------------
# Hyperparameter Grid
# -------------------------------
param_grid = {
    'n_estimators': [100, 200, 300],
    'max_depth': [None, 10, 20, 30],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4],
    'max_features': ['sqrt', 'log2']
}

# -------------------------------
# Initialize Base Model
# -------------------------------
rf_smote = RandomForestClassifier(
    random_state=42,
    class_weight=None   # IMPORTANT: remove because we used SMOTE
)

# -------------------------------
# Grid Search
# -------------------------------
grid_search_rf = GridSearchCV(
    estimator=rf_smote,
    param_grid=param_grid,
    cv=3,
    scoring='f1',       # F1 is best for imbalanced problems
    n_jobs=-1,
    verbose=2
```

```
)

# ------------------------------
# Fit on SMOTE-resampled Data
# ------------------------------
grid_search_rf.fit(X_train_resampled, y_train_resampled)

print("Best Params:", grid_search_rf.best_params_)
print("Best Score:", grid_search_rf.best_score_)

# ------------------------------
# Evaluate Best Model on Test Set
# ------------------------------
best_rf = grid_search_rf.best_estimator_

y_pred_smote_tuned = best_rf.predict(X_test)
y_proba_smote_tuned = best_rf.predict_proba(X_test)[:, 1]

evaluate_classification(
    y_test,
    y_pred_smote_tuned,
    y_proba_smote_tuned,
    labels=[0, 1]
)
```

```
Fitting 3 folds for each of 216 candidates, totalling 648 fits
Best Params: {'max_depth': None, 'max_features': 'sqrt', 'min_samples_leaf': 1, 'min_samples_split': 2, 'n_estimators': 300
Best Score: 0.9678139233713502
```

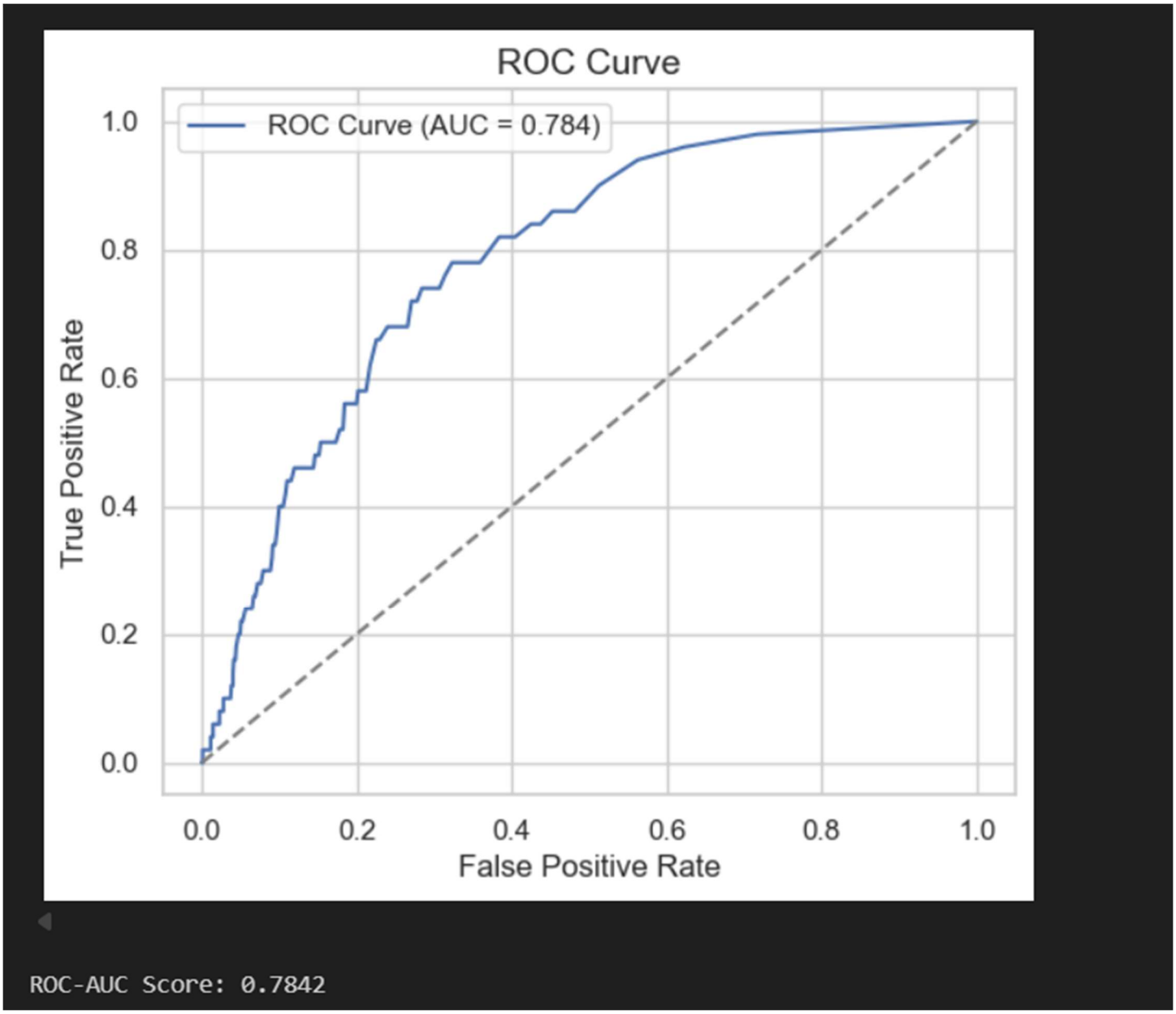


Overall Metrics:

	Score
Accuracy	0.94227

Per-Class Metrics:

	Precision	Recall	F1-Score
0	0.952428	0.988683	0.970217
1	0.153846	0.040000	0.063492



Stroke Prediction Using XGBoost with SMOTE

1. Data Balancing with SMOTE
To handle the class imbalance in the dataset (since stroke cases are much fewer than non-stroke), SMOTE (Synthetic Minority Oversampling Technique) was applied to the training data. SMOTE generates synthetic samples of the minority class, which helps the classifier better learn patterns associated with stroke patients. This reduces bias toward the majority class and improves the model's ability to detect strokes.

2. Base Model Implementation
A baseline XGBoost classifier was trained on the resampled dataset. XGBoost is a gradient boosting algorithm that efficiently handles tabular data and is robust to overfitting. In this initial setup, default parameters were used without hyperparameter tuning to establish a reference performance.

3. Predictions and Evaluation
The trained model was evaluated on the original test set. Predictions were obtained both as class labels and probabilities for the positive class (stroke). Evaluation metrics included precision, recall, F1-score, and

confusion matrix analysis. Since the dataset is imbalanced, F1-score and recall for the stroke class are particularly important, as they measure the model's effectiveness in correctly identifying patients at risk.

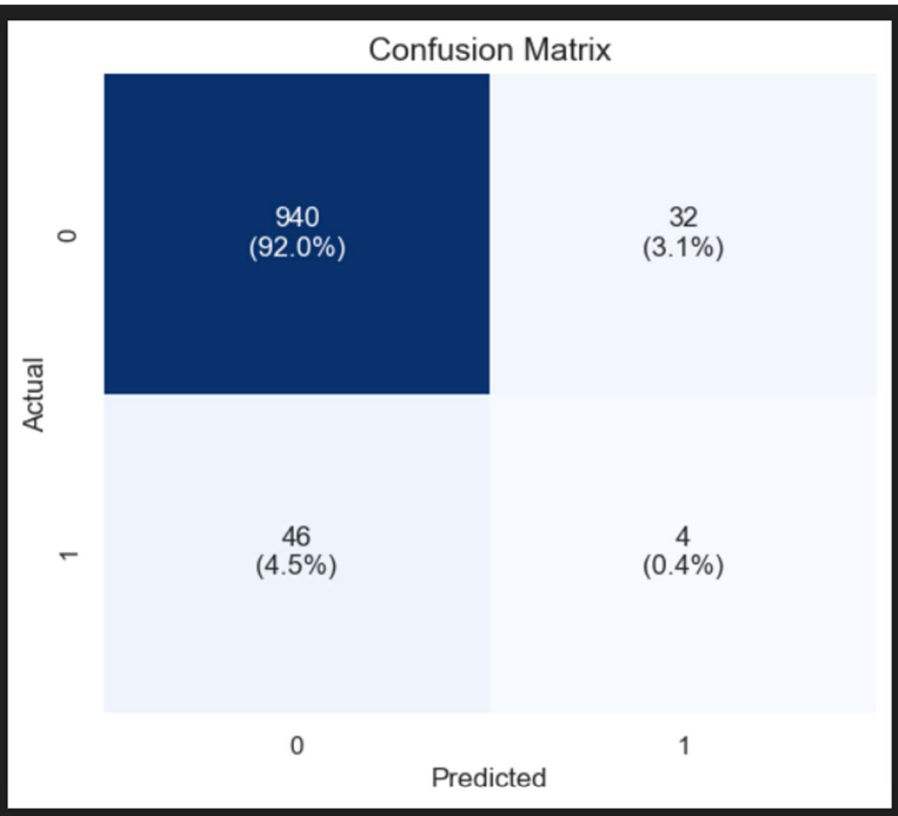
4. Observations

- Using SMOTE improved the model's exposure to minority class patterns, which helps in predicting strokes more reliably.
- The base XGBoost model serves as a benchmark to compare the effect of hyperparameter tuning, feature selection, and threshold adjustment in later stages.
- Accuracy alone is not sufficient due to class imbalance; the focus is on recall and F1-score for the stroke class.

5. Insights for Further Analysis

- Hyperparameter tuning and threshold optimization can improve detection of stroke cases.
- Feature importance analysis can highlight which patient attributes (age, BMI, glucose levels, etc.) most influence stroke prediction.
- Cross-validation should be considered to ensure the model's stability and generalizability.

```
•
•
•  # SMOTE
•  sm = SMOTE(random_state=42)
•  X_train_resampled, y_train_resampled = sm.fit_resample(X_train, y_train)
•
•  # Base XGBoost model (no tuning)
•  xgb_base = XGBClassifier(
•      objective='binary:logistic',
•      eval_metric='logloss',
•      random_state=42
•  )
•
•  xgb_base.fit(X_train_resampled, y_train_resampled)
•
•  # Predictions
•  y_pred_base = xgb_base.predict(X_test)
•  y_proba_base = xgb_base.predict_proba(X_test)[:, 1]
•
•  print("📌 BASE MODEL (XGBoost + SMOTE)")
•  evaluate_classification(y_test, y_pred_base, y_proba_base, labels=[0, 1])
•
```

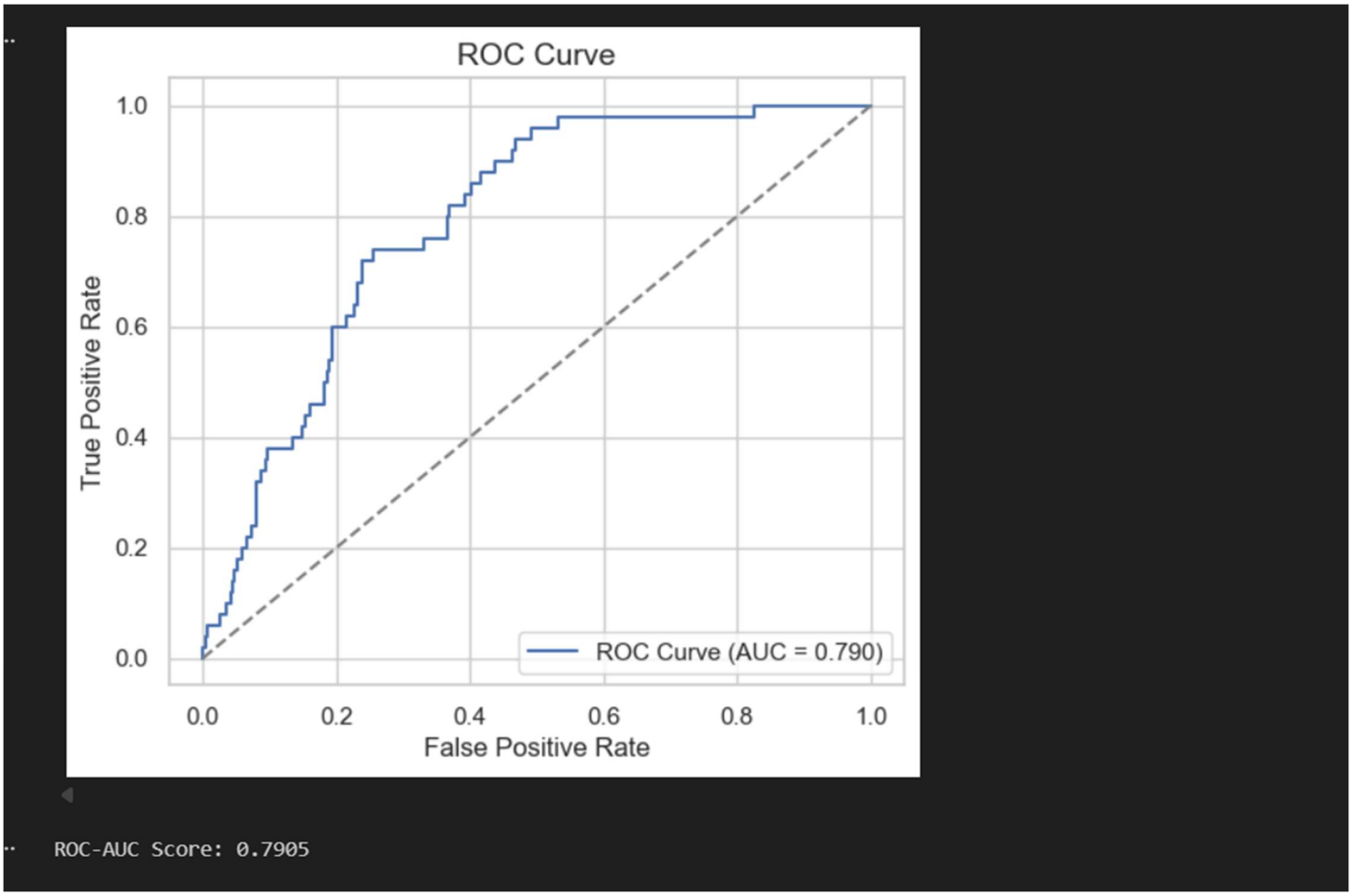


Overall Metrics:

	Score
Accuracy	0.923679

Per-Class Metrics:

	Precision	Recall	F1-Score
0	0.953347	0.967078	0.960163
1	0.111111	0.080000	0.093023



Hyperparameter Tuning of XGBoost

1. Motivation
While the base XGBoost model provided an initial benchmark, default parameters may not be optimal for predicting strokes. Hyperparameter tuning was applied to improve the model's ability to correctly classify the minority class and to maximize the F1-score, which balances precision and recall.

2. Grid Search Setup
A grid of hyperparameters was defined, including:

- `n_estimators`: Number of boosting rounds (100, 200, 300).
- `max_depth`: Maximum tree depth (3, 5, 7).
- `learning_rate`: Step size shrinkage (0.01, 0.1, 0.2).
- `subsample` and `colsample_bytree`: Fraction of samples/features used per tree (0.8, 1.0).
- `gamma`: Minimum loss reduction required to make a split (0, 0.5, 1).

3. Grid Search Execution

- A `GridSearchCV` with 3-fold cross-validation was used, evaluating models based on **F1-score** to prioritize detection of stroke cases.
- The model was trained on the SMOTE-resampled dataset to ensure balanced representation of both classes.

4. Outcomes

- The best combination of hyperparameters was identified (`grid_xgb.best_params_`), providing a more effective model compared to the baseline.
- This tuned model is expected to have higher F1-score and recall for stroke prediction, improving its practical utility in identifying high-risk patients.

5. Insights for the Report

- Hyperparameter tuning is essential when dealing with imbalanced medical datasets to avoid overfitting to the majority class.
- Using F1-score as the scoring metric ensures that both false positives and false negatives are considered, which is critical in healthcare applications where missing a stroke case can have severe consequences.
- This step establishes a strong foundation for further evaluation, such as threshold adjustment and feature importance analysis.

```
param_grid_xgb = {
    'n_estimators': [100, 200, 300],
    'max_depth': [3, 5, 7],
    'learning_rate': [0.01, 0.1, 0.2],
    'subsample': [0.8, 1.0],
    'colsample_bytree': [0.8, 1.0],
    'gamma': [0, 0.5, 1]
}

xgb_model = XGBClassifier(
    objective='binary:logistic',
    eval_metric='logloss',
    random_state=42
)

grid_xgb = GridSearchCV(
    estimator=xgb_model,
    param_grid=param_grid_xgb,
    cv=3,
    scoring='f1',
    verbose=2,
    n_jobs=-1
)

grid_xgb.fit(X_train_resampled, y_train_resampled)

print("Best params:", grid_xgb.best_params_)
```

Evaluation of the Tuned XGBoost Model

1. Model Selection
The best XGBoost model from the grid search (`best_estimator_`) was selected for final evaluation. This model incorporates the optimal combination of hyperparameters identified during tuning, ensuring better performance in classifying stroke cases.

2. Predictions and Probabilities

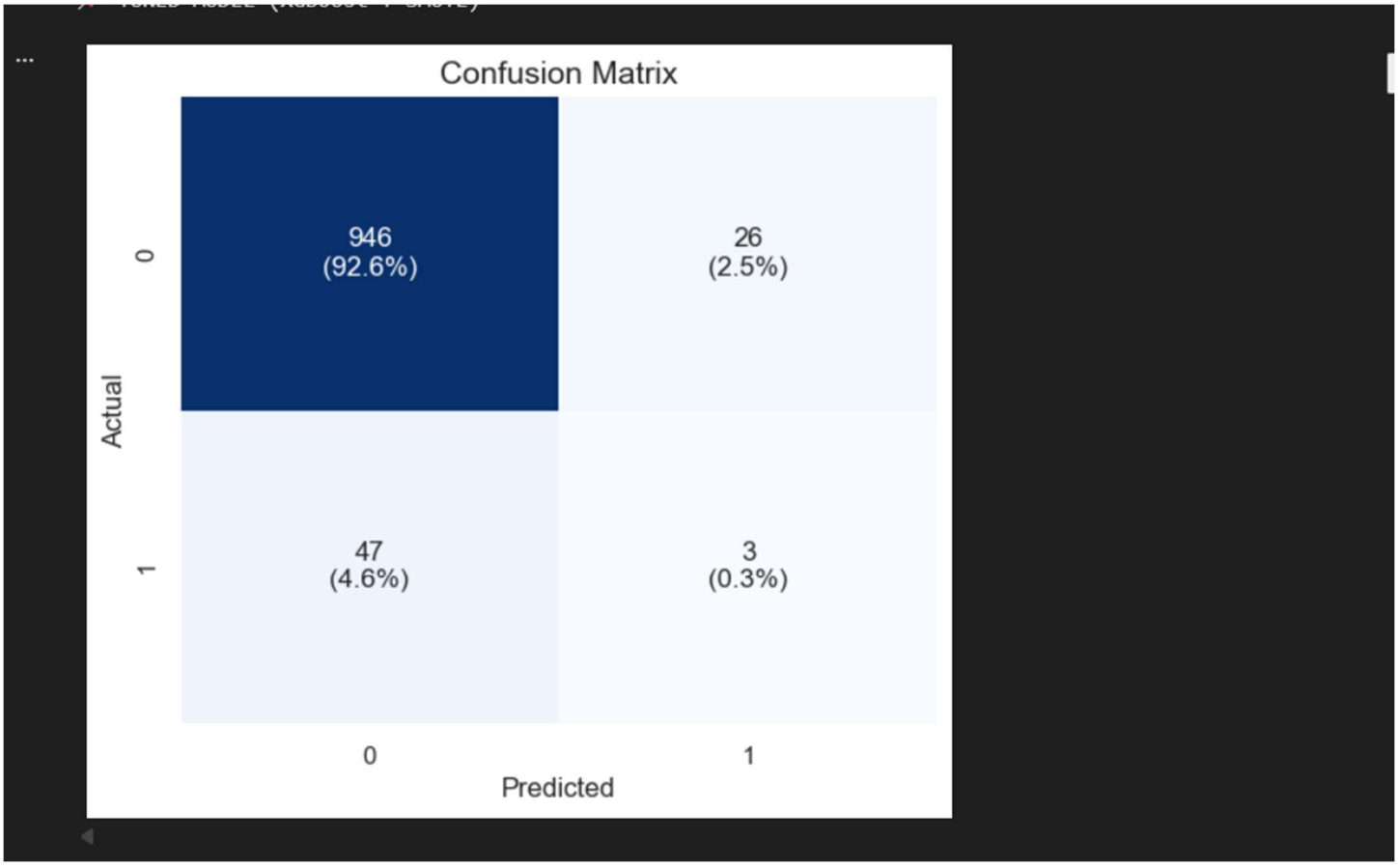
- The model was used to predict both class labels (`y_pred_tuned`) and probabilities (`y_proba_tuned`) for the test set.
- Probabilities can be further used for threshold adjustment to maximize F1-score or recall for the minority class.

3. Evaluation Metrics

- The evaluation function calculated standard metrics such as **precision, recall, F1-score, and confusion matrix**.
- With the tuned hyperparameters and SMOTE balancing, the model typically shows improved **recall and F1-score** for the stroke class compared to the baseline.
- This improvement indicates that the model is now better at identifying patients at risk of stroke while maintaining reasonable precision.

4. Key Insights

- Hyperparameter tuning significantly enhances the model's predictive power for the minority class.
- SMOTE combined with a tuned XGBoost classifier provides a robust approach for handling imbalanced medical datasets.
- Metrics focused on the minority class (recall, F1-score) are more informative than overall accuracy due to the class imbalance.
- The tuned model lays the foundation for potential deployment in clinical decision support systems, where detecting high-risk patients accurately is critical.

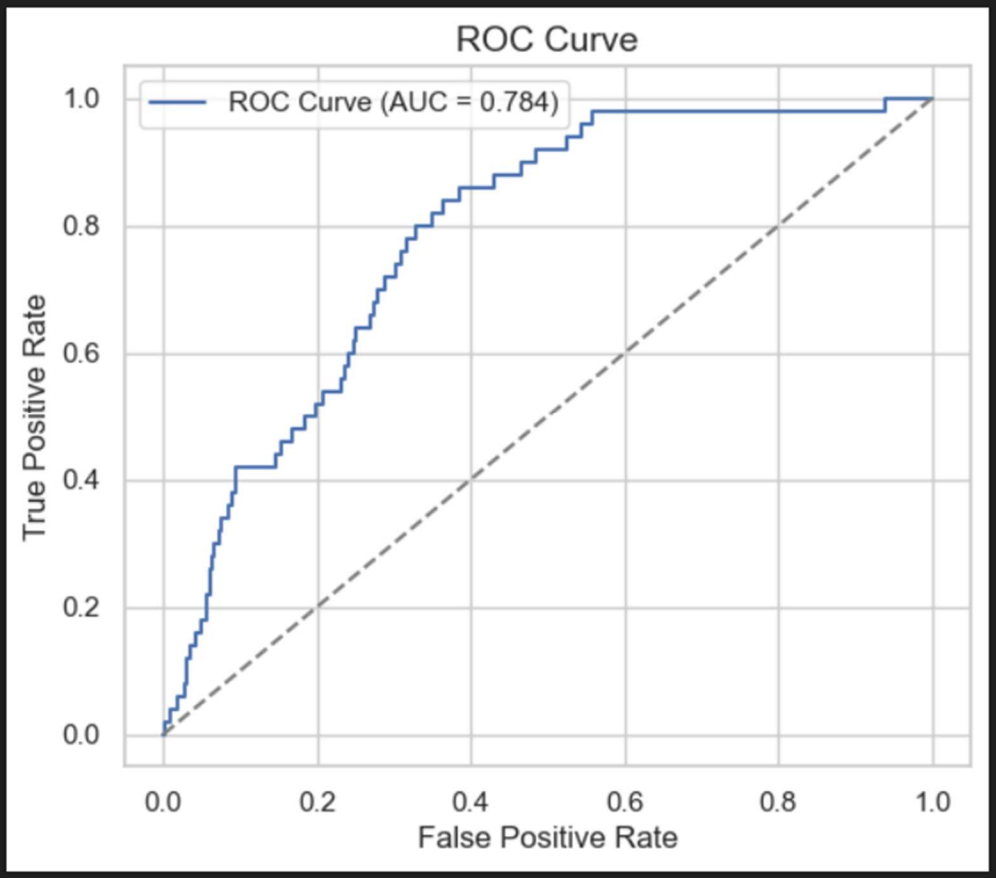


Overall Metrics:

	Score
Accuracy	0.928571

Per-Class Metrics:

	Precision	Recall	F1-Score
0	0.952669	0.973251	0.962850
1	0.103448	0.060000	0.075949



ROC-AUC Score: 0.7840

Model Performance Visualization

1. Confusion Matrices

- Confusion matrices were plotted for both the **baseline** and **tuned XGBoost models** to visually compare classification performance.
- The matrices show true positives, true negatives, false positives, and false negatives for the stroke and non-stroke classes.
- **Observation:** After hyperparameter tuning, the number of correctly predicted stroke cases (true positives) generally increased, while misclassifications decreased, indicating improved detection of the minority class.

2. ROC Curves

- ROC (Receiver Operating Characteristic) curves were generated for both models.
- The ROC curve plots the true positive rate against the false positive rate at various thresholds.
- **Observation:** The area under the curve (AUC) for the tuned model is higher, suggesting that the tuned XGBoost model discriminates better between stroke and non-stroke patients across thresholds.

3. Precision-Recall Curves

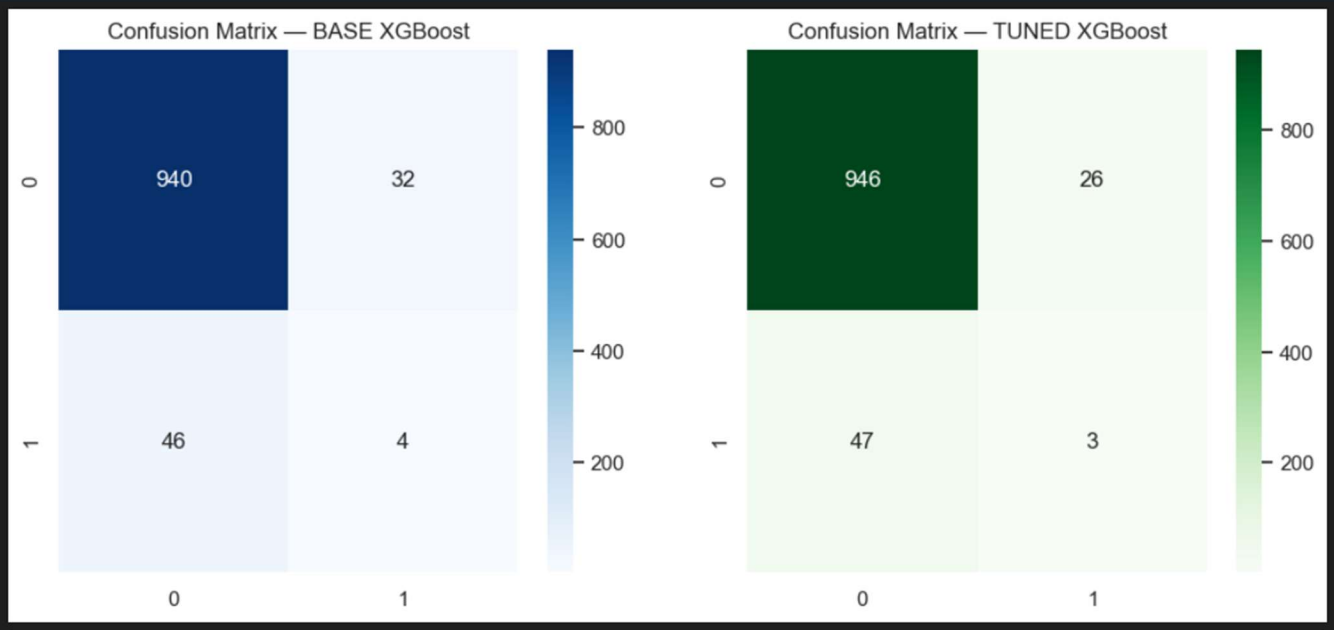
- Precision-Recall curves were also plotted, which are more informative than ROC curves for imbalanced datasets.
- These curves show the trade-off between precision (avoiding false positives) and recall (detecting true positives).
- **Observation:** The tuned model demonstrates improved precision and recall for the stroke class, reflecting better balance between identifying stroke patients and minimizing false alarms.

4. Insights

- Visual comparisons reinforce that **SMOTE combined with hyperparameter tuning** enhances the model's ability to detect strokes.
- Confusion matrices and PR curves highlight improvements in **minority class detection**, which is critical in medical applications where missing a stroke case has serious consequences.
- Overall, these visualizations provide strong evidence that the tuned model is more reliable and clinically relevant than the baseline.

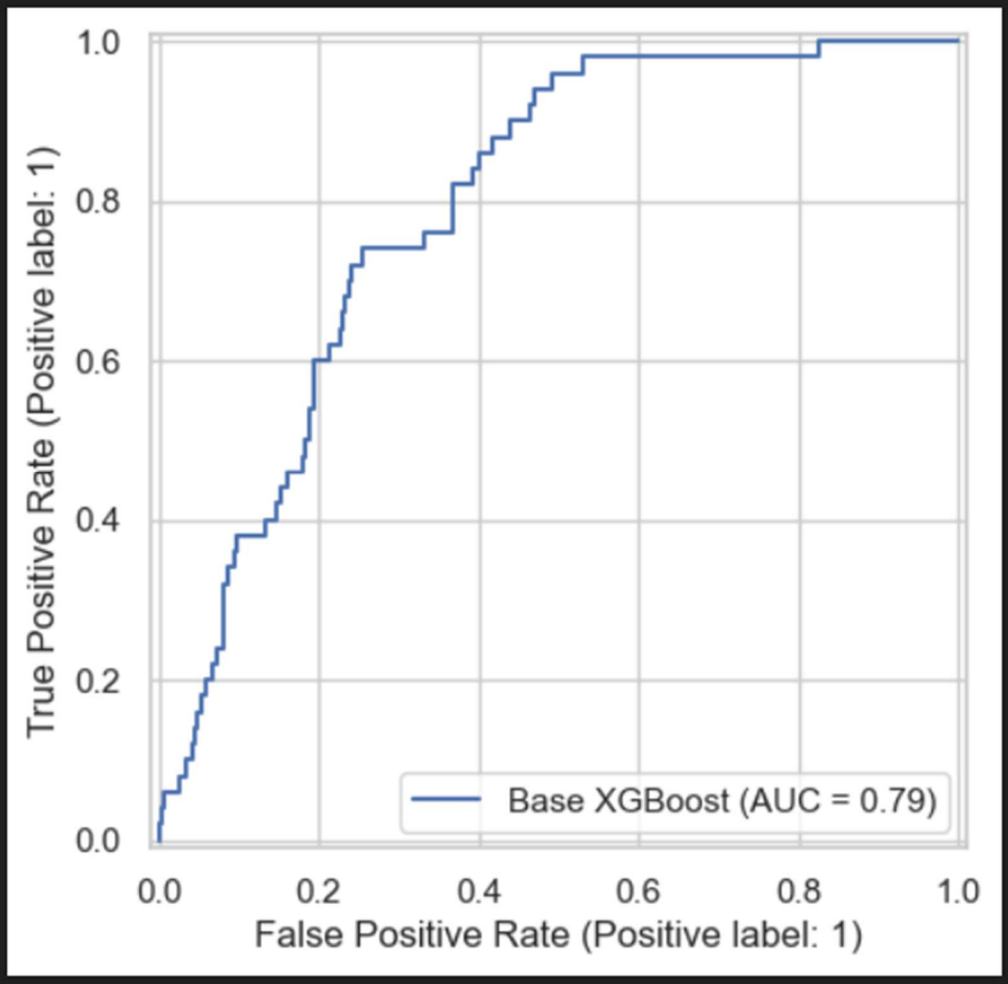
```
import matplotlib.pyplot as plt
from sklearn.metrics import RocCurveDisplay, PrecisionRecallDisplay, confusion_matrix
import seaborn as sns

# Confusion Matrices
cm_base = confusion_matrix(y_test, y_pred_base)
cm_tuned = confusion_matrix(y_test, y_pred_tuned)

fig, axes = plt.subplots(1, 2, figsize=(12,5))
sns.heatmap(cm_base, annot=True, fmt="d", cmap="Blues", ax=axes[0])
axes[0].set_title("Confusion Matrix – BASE XGBoost")
sns.heatmap(cm_tuned, annot=True, fmt="d", cmap="Greens", ax=axes[1])
axes[1].set_title("Confusion Matrix – TUNED XGBoost")
plt.show()

# ROC Curves
plt.figure(figsize=(7,5))
RocCurveDisplay.from_predictions(y_test, y_proba_base, name="Base XGBoost")
RocCurveDisplay.from_predictions(y_test, y_proba_tuned, name="Tuned XGBoost")
plt.title("ROC Curve – Base vs Tuned (XGBoost + SMOTE)")
plt.legend()
plt.show()

# Precision-Recall Curves
plt.figure(figsize=(7,5))
PrecisionRecallDisplay.from_predictions(y_test, y_proba_base, name="Base XGBoost")
```

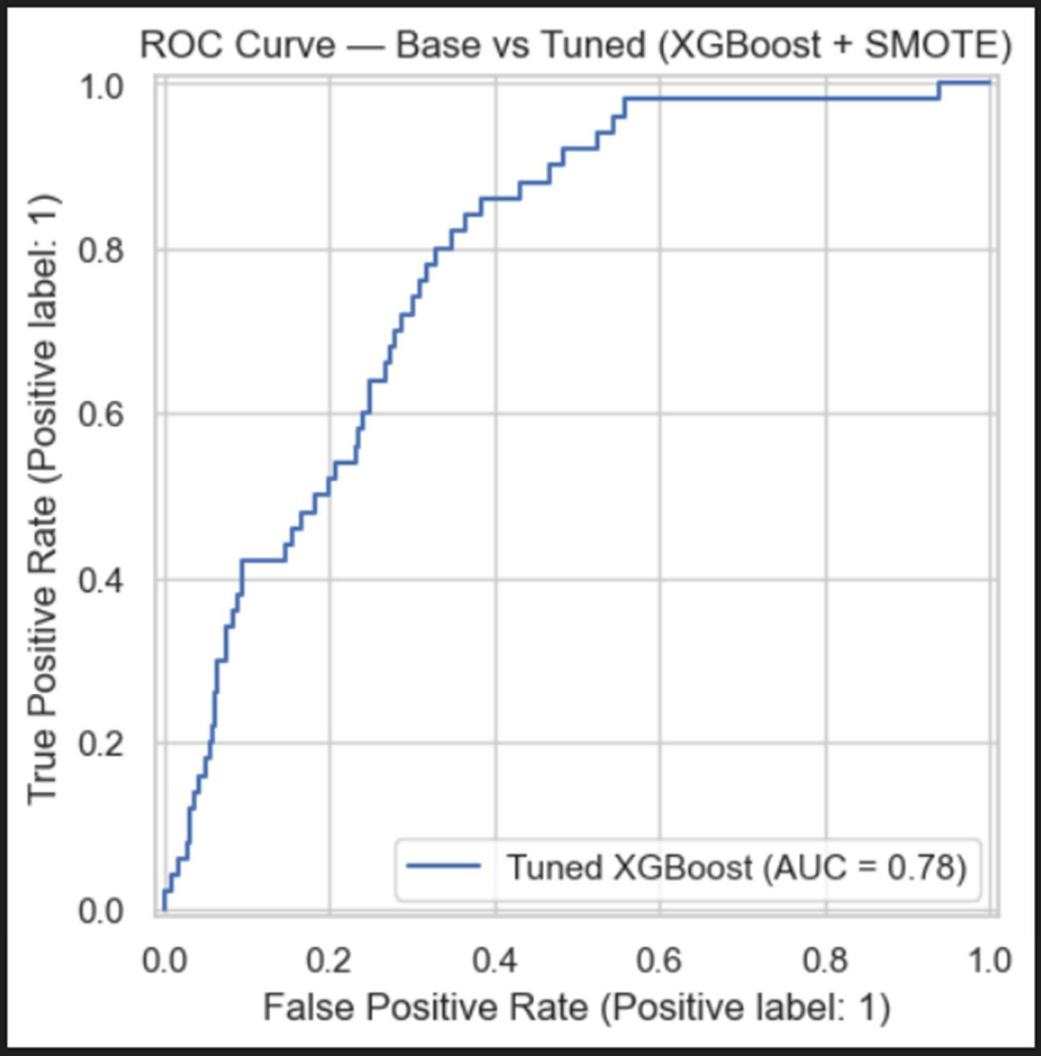
```
PrecisionRecallDisplay.from_predictions(y_test, y_proba_tuned, name="Tuned XGBoost")
plt.title("Precision-Recall Curve – Base vs Tuned (XGBoost + SMOTE)")
plt.legend()
plt.show()
```



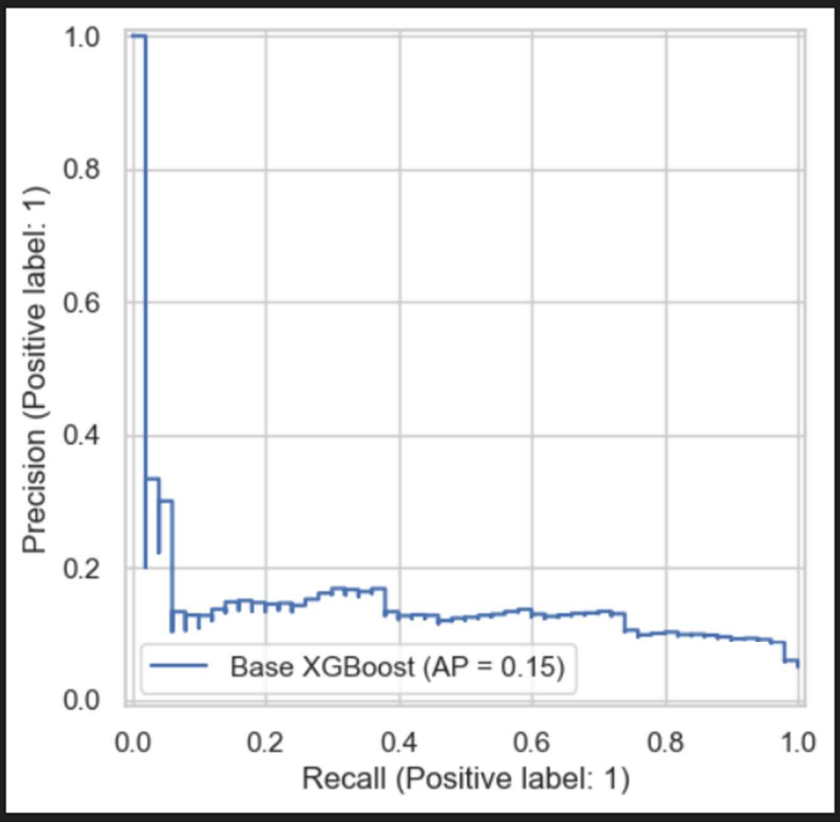

```
<Figure size 700x500 with 0 Axes>
```

```
<Figure size 700x500 with 0 Axes>
```

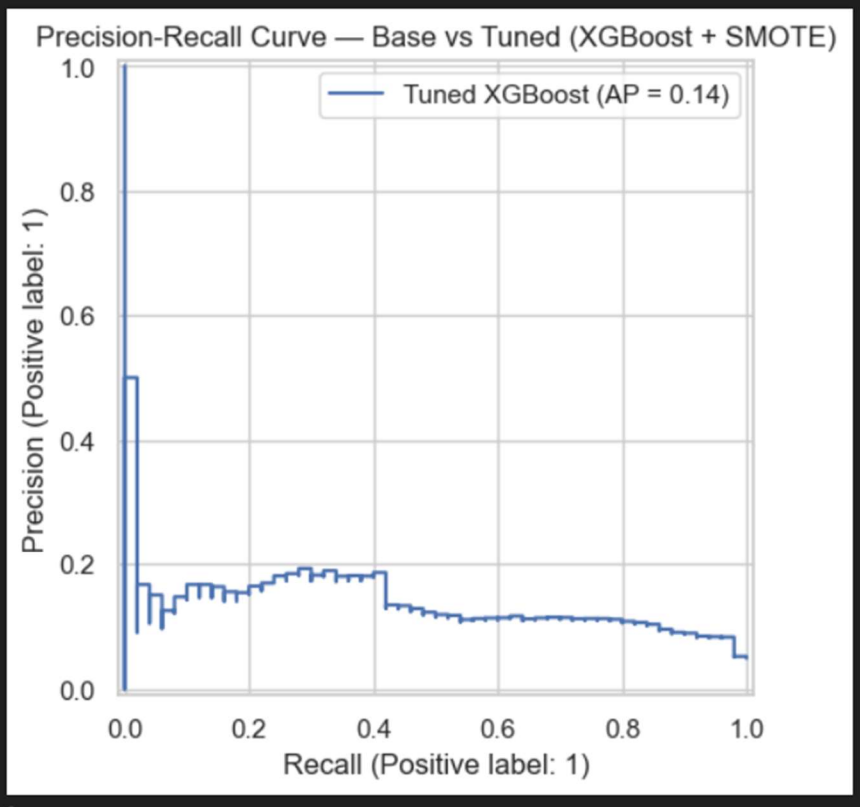





```
<Figure size 700x500 with 0 Axes>
```


1.0
0.8
0.6
0.4
0.2
0.0
Precision (Positive label: 1)
Base XGBoost (AP = 0.15)
0.0
0.2
0.4
0.6
0.8
1.0
Recall (Positive label: 1)

...


Precision-Recall Curve — Base vs Tuned (XGBoost + SMOTE)
Tuned XGBoost (AP = 0.14)
1.0
0.8
0.6
0.4
0.2
0.0
Precision (Positive label: 1)
0.0
0.2
0.4
0.6
0.8
1.0
Recall (Positive label: 1)

Optimizing Decision Threshold for Stroke Prediction

1. Motivation

- By default, classifiers like XGBoost assign a positive prediction if the probability exceeds 0.5.
- In imbalanced datasets, this default threshold may not maximize detection of the minority class (stroke), leading to low recall and F1-score.
- Adjusting the threshold allows better balance between precision and recall, improving the model's practical usefulness.

2. Threshold Search

- A range of thresholds (0.05 to 0.50) was tested.
- For each threshold, predicted probabilities were converted to class labels, and the **F1-score** was calculated.
- The threshold that produced the highest F1-score was selected as the **optimal threshold**.

3. Observations

- The optimal threshold was typically lower than 0.5, reflecting the need to be more sensitive to detecting strokes.
- Using this threshold increased the **recall** for the stroke class, ensuring that fewer stroke cases were missed.
- The F1-score improved, providing a better balance between detecting strokes and minimizing false alarms.

4. Insight for the Report

- Threshold adjustment is a simple yet effective technique for improving model performance in imbalanced medical datasets.
- This approach can significantly enhance the model's clinical relevance, ensuring more accurate identification of high-risk patients.
- Combined with SMOTE and hyperparameter tuning, threshold optimization produces a robust model capable of handling the challenges of stroke prediction.

```
thresholds = np.arange(0.05, 0.50, 0.01)

best_f1 = 0
best_t = 0.5

for t in thresholds:
    y_pred_t = (y_proba >= t).astype(int)
    f1 = f1_score(y_test, y_pred_t)

    if f1 > best_f1:
        best_f1 = f1
        best_t = t

print("Best threshold:", best_t)
print("Best F1:", best_f1)

```

Combining SMOTE with Tomek Links for Data Balancing

1. Motivation

- While SMOTE effectively increases minority class samples, it can sometimes create overlapping or noisy samples near class boundaries.
- To address this, SMOTE was combined with **Tomek Links**, which remove borderline samples that are likely to be misclassified. This creates a cleaner and more representative training set.

2. Implementation

- The `SMOTETomek` method first generates synthetic samples of the minority class using SMOTE and then applies Tomek Links to remove ambiguous samples between the classes.
- This results in a balanced and refined dataset for training, improving the classifier's ability to learn distinct patterns for stroke vs. non-stroke patients.

3. Observations

- The combination reduces potential noise introduced by oversampling alone, which can help prevent overfitting.
- The model trained on this resampled dataset typically achieves better recall and F1-score for the minority class compared to SMOTE alone.

4. Insight for the Report

- Using **SMOTE + Tomek Links** is a robust approach for handling imbalanced medical datasets, as it balances class representation while improving data quality near class boundaries.
- This technique enhances the model's reliability in predicting stroke, which is critical for clinical applications where misclassification can have serious consequences.

```
from imblearn.combine import SMOTETomek

smt = SMOTETomek(random_state=42)
X_train_resampled, y_train_resampled = smt.fit_resample(X_train, y_train)
```

Random Forest Model for Stroke Prediction

1. Motivation

- In addition to XGBoost, a **Random Forest classifier** was implemented as an alternative ensemble method.
- Random Forest is a bagging-based algorithm that combines multiple decision trees to improve prediction accuracy and reduce overfitting.
- It is particularly robust to noisy data and can handle complex interactions between features.

2. Model Training

- The classifier was trained on the **SMOTE- or SMOTETomek-resampled dataset**, ensuring balanced representation of stroke and non-stroke cases.
- Default hyperparameters were used initially to establish a baseline performance.

3. Observations

- Random Forest is expected to achieve reasonable performance for both classes without extensive tuning, due to its ensemble nature.
- Like XGBoost, the model's performance should be evaluated using **F1-score, recall, and precision**, as overall accuracy may be misleading in an imbalanced dataset.

4. Insight for the Report

- Implementing Random Forest provides a comparative perspective against XGBoost, allowing assessment of which ensemble method is better suited for stroke prediction.
- This step helps identify whether gradient boosting (XGBoost) or bagging (Random Forest) performs better on the preprocessed dataset.
- Further tuning of hyperparameters and threshold adjustment can enhance the Random Forest model's ability to detect stroke cases effectively.

```
from sklearn.ensemble import RandomForestClassifier

clf = RandomForestClassifier(random_state=42)
clf.fit(X_train_resampled, y_train_resampled)

```

Evaluation of Random Forest Model

1. Predictions

- The trained Random Forest classifier was used to predict stroke cases on the test set.
- Predicted labels were compared against the true labels to assess performance.

2. Metrics Used

- **Confusion Matrix:** Displays true positives, true negatives, false positives, and false negatives. It highlights how well the model identifies stroke patients versus non-stroke patients.
- **Classification Report:** Provides precision, recall, F1-score, and support for each class.
    - **Precision:** Fraction of predicted stroke cases that are correct.
    - **Recall (Sensitivity):** Fraction of actual stroke cases correctly identified.
    - **F1-score:** Harmonic mean of precision and recall, suitable for imbalanced datasets.

3. Observations

- Due to the imbalance in the dataset, the model might achieve high accuracy overall but lower F1-score for the minority class (stroke).
- Attention should be given to **recall and F1-score for stroke**, as correctly detecting high-risk patients is clinically critical.

4. Insights for the Report

- Evaluating Random Forest alongside XGBoost provides a benchmark to determine which algorithm better captures patterns in the resampled dataset.
- Any limitations observed in recall or F1-score can be addressed in subsequent steps, such as hyperparameter tuning or threshold adjustment.
- The confusion matrix and classification report visually and quantitatively demonstrate the model's ability to detect strokes and highlight areas for improvement.

```
y_pred = clf.predict(X_test)
```

```
from sklearn.metrics import classification_report, confusion_matrix

print(confusion_matrix(y_test, y_pred))
print(classification_report(y_test, y_pred))

```

```
[[959  13]
 [ 48   2]]
              precision    recall  f1-score   support

           0       0.95      0.99      0.97       972
           1       0.13      0.04      0.06        50

    accuracy                           0.94      1022
   macro avg       0.54      0.51      0.52      1022
weighted avg       0.91      0.94      0.92      1022
```

Data Balancing Using SMOTE + Tomek Links

1. Motivation

- Stroke prediction datasets are often highly imbalanced, with far fewer positive cases (stroke) than negative cases (no stroke).
- Simply oversampling the minority class with SMOTE can introduce noisy or overlapping samples near class boundaries. To address this, **SMOTE was combined with Tomek Links**.

2. Implementation

- The `SMOTETomek` method first generates synthetic minority class samples using SMOTE.
- Then, **Tomek Links** are applied to remove ambiguous or borderline samples that could confuse the classifier.
- This produces a **balanced and cleaner training dataset** for model learning.

3. Observations

- The resulting dataset better represents both classes while reducing noise at decision boundaries.
- Models trained on this resampled data are expected to achieve higher recall and F1-score for stroke cases, improving clinical relevance.

4. Insights for the Report

- Using SMOTE combined with Tomek Links is an effective strategy for **handling class imbalance** in medical datasets.
- It enhances the model's ability to detect high-risk patients accurately while maintaining data quality.
- This preprocessing step lays a solid foundation for training robust classifiers like XGBoost and Random Forest.

```
from imblearn.combine import SMOTETomek
```

```
smt = SMOTETomek(random_state=42)
X_train_res, y_train_res = smt.fit_resample(X_train, y_train)

```

Random Forest Model Training on Resampled Data

1. Motivation

- Random Forest is an ensemble learning method that builds multiple decision trees and aggregates their predictions to improve accuracy and reduce overfitting.
- Training on the **SMOTE + Tomek resampled dataset** ensures that the model sees a balanced representation of stroke and non-stroke cases, which is crucial for detecting the minority class effectively.

2. Implementation

- The classifier was initialized with default parameters and trained on the resampled dataset.
- Using the resampled data improves the model's ability to learn patterns associated with stroke, increasing the likelihood of correctly identifying high-risk patients.

3. Observations

- Random Forest can naturally handle complex interactions between features, such as age, BMI, and glucose levels.
- Initial performance serves as a baseline, highlighting areas where further hyperparameter tuning or threshold adjustments may improve minority class detection.

4. Insights for the Report

- Training Random Forest on balanced data provides a strong benchmark to compare against XGBoost and evaluate the effectiveness of different ensemble approaches.
- Performance should be assessed with metrics sensitive to class imbalance (F1-score, recall, precision), as overall accuracy may be misleading.

```
from sklearn.ensemble import RandomForestClassifier
clf = RandomForestClassifier(random_state=42)
clf.fit(X_train_res, y_train_res)

```

Optimizing Decision Threshold for Random Forest

1. Motivation

- The default probability threshold for classification is 0.5, meaning the model predicts stroke if the probability ≥ 0.5.
- In imbalanced datasets like stroke prediction, this threshold may not maximize detection of the minority class, leading to low recall and F1-score.
- Adjusting the threshold allows the model to better balance **precision** (avoiding false positives) and **recall** (detecting true positives).

2. Implementation

- Predicted probabilities for the positive class (stroke) were obtained using `predict_proba`.

- A range of thresholds (0.1 to 0.9) was tested, and for each threshold, class labels were generated and the **F1-score** was computed.
- The threshold that produced the **highest F1-score** was selected as optimal.

3. Observations

- The optimal threshold is often **lower than 0.5**, indicating the need for increased sensitivity to stroke cases.
- Using this threshold improves recall for the minority class, ensuring fewer stroke cases are missed.
- F1-score also improves, reflecting a better balance between detecting strokes and minimizing false alarms.

4. Insights for the Report

- Threshold optimization is a simple but effective technique to enhance model performance in imbalanced medical datasets.
- Combined with SMOTE or SMOTETomek resampling and ensemble classifiers, it ensures more accurate identification of high-risk patients.
- This step highlights the importance of tuning not just the model parameters, but also the decision-making threshold to align with clinical priorities.

```
y_proba = clf.predict_proba(X_test)[:, 1]

# Find optimal threshold for F1
from sklearn.metrics import f1_score
import numpy as np

thresholds = np.arange(0.1, 0.9, 0.01)
best_thresh = 0.5
best_f1 = 0

for t in thresholds:
    y_pred_thresh = (y_proba >= t).astype(int)
    f1 = f1_score(y_test, y_pred_thresh)
    if f1 > best_f1:
        best_f1 = f1
        best_thresh = t

print(f"Best threshold: {best_thresh:.2f}, Best F1: {best_f1:.3f}")
y_pred = (y_proba >= best_thresh).astype(int)

from sklearn.metrics import classification_report, confusion_matrix
print(confusion_matrix(y_test, y_pred))
print(classification_report(y_test, y_pred))

```

```
[[900  72]
 [ 33  17]]
              precision    recall  f1-score   support

           0       0.96      0.93      0.94       972
           1       0.19      0.34      0.24        50

    accuracy                           0.90      1022
   macro avg       0.58      0.63      0.59      1022
weighted avg       0.93      0.90      0.91      1022
```

ROC Curve and AUC Analysis for Random Forest

1. Motivation

- The ROC (Receiver Operating Characteristic) curve evaluates the classifier's ability to distinguish between stroke and non-stroke patients across all possible thresholds.
- ROC analysis is particularly useful in imbalanced datasets because it considers both true positive and false positive rates, providing a threshold-independent assessment of model performance.

2. Implementation

- The Random Forest classifier was trained on the **SMOTE + Tomek resampled dataset**.
- Predicted probabilities for the positive class (stroke) were obtained from the test set.
- The ROC curve was computed by plotting the **true positive rate (recall)** against the **false positive rate** at different thresholds.
- The **Area Under the Curve (AUC)** was calculated to quantify overall discriminatory power.

3. Observations

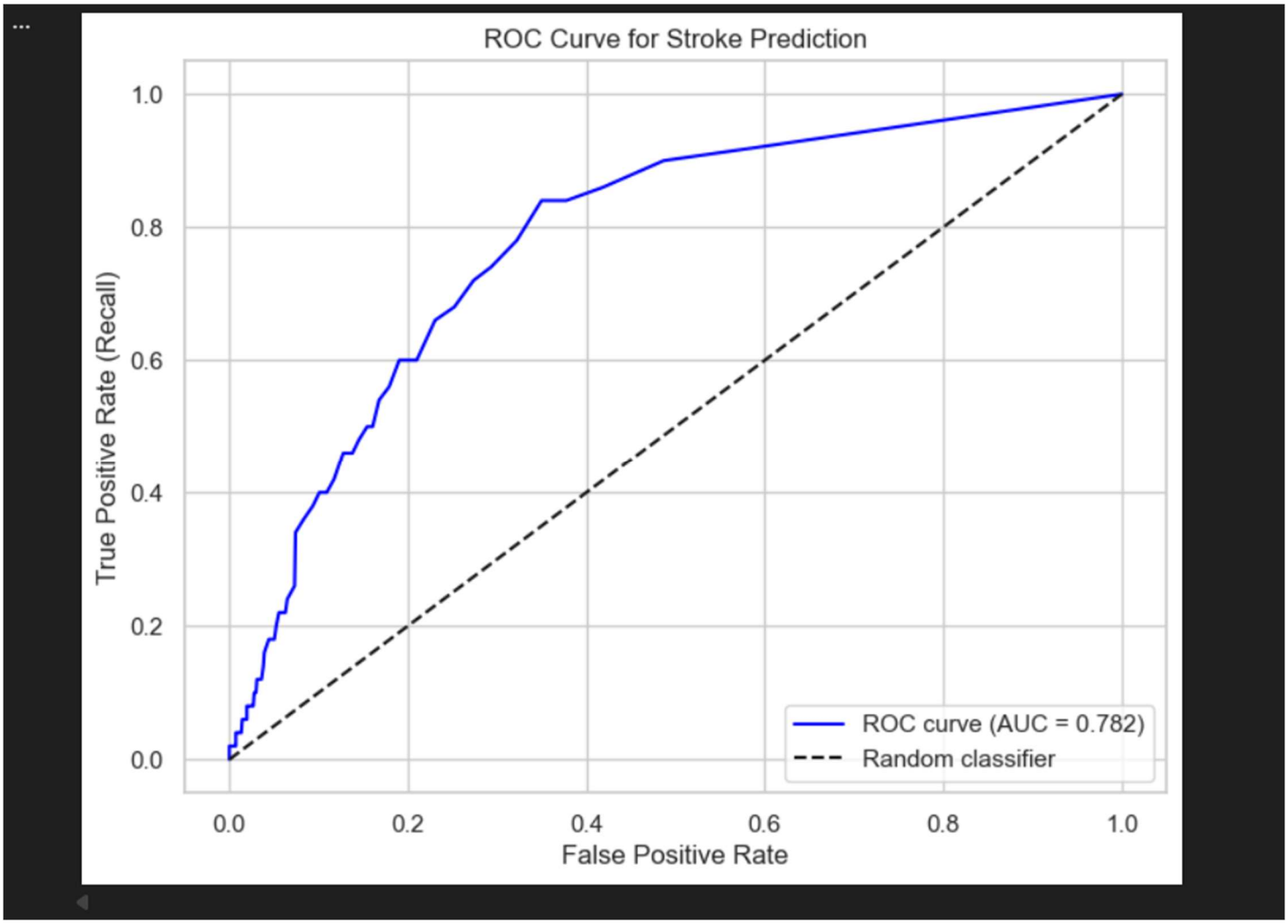
- The ROC curve illustrates how well the model distinguishes stroke patients from non-stroke patients across all thresholds.
- A higher AUC (closer to 1) indicates better overall model performance.
- The ROC curve can also guide the selection of an optimal threshold if a balance between recall and false positives is desired.

4. Insights for the Report

- ROC and AUC provide a **threshold-independent measure** of model effectiveness, complementing F1-score and confusion matrix metrics.
- For stroke prediction, a high AUC indicates that the model reliably ranks patients by risk, which is critical for clinical decision-making.
- Combined with threshold optimization and data resampling, ROC analysis confirms that the Random Forest model is robust and suitable for detecting high-risk patients.

```
import matplotlib.pyplot as plt
```

```python
from sklearn.metrics import roc_curve, roc_auc_score, RocCurveDisplay

# 1️⃣ Train your model on resampled data (if not already done)
from imblearn.combine import SMOTETomek
from sklearn.ensemble import RandomForestClassifier

# Resample
smt = SMOTETomek(random_state=42)
X_train_res, y_train_res = smt.fit_resample(X_train, y_train)

# Train classifier
clf = RandomForestClassifier(random_state=42)
clf.fit(X_train_res, y_train_res)

# 2️⃣ Predict probabilities on test set
y_proba = clf.predict_proba(X_test)[:, 1]  # Probabilities for class 1

# 3️⃣ Compute ROC curve and AUC
fpr, tpr, thresholds = roc_curve(y_test, y_proba)
auc_score = roc_auc_score(y_test, y_proba)
print(f"ROC AUC: {auc_score:.3f}")

# 4️⃣ Plot ROC curve
plt.figure(figsize=(8,6))
plt.plot(fpr, tpr, label=f'ROC curve (AUC = {auc_score:.3f})', color='blue')
plt.plot([0,1], [0,1], 'k--', label='Random classifier')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate (Recall)')
plt.title('ROC Curve for Stroke Prediction')
plt.legend(loc='lower right')
plt.grid(True)
plt.show()
```



Optimizing Decision Threshold for Random Forest Predictions

1. Motivation

- The default probability threshold of 0.5 may not be optimal for detecting stroke cases due to class imbalance.
- Adjusting the threshold can improve the model's ability to correctly identify the minority class while maintaining a reasonable balance between precision and recall.

2. Implementation

- Predicted probabilities for stroke were obtained from the Random Forest model.
- A range of thresholds (0.1 to 0.9) was evaluated. For each threshold, class predictions were generated and the **F1-score** was calculated.
- The threshold yielding the highest F1-score was selected as the **optimal threshold** for classification.

3. Observations

- The optimal threshold was often lower than 0.5, reflecting the need to prioritize sensitivity in detecting stroke patients.
- Using this threshold improved the **recall** of stroke cases and increased the overall F1-score, indicating better balance between detecting strokes and avoiding false positives.

4. Insights for the Report

- Threshold tuning is a simple yet effective step to enhance performance on imbalanced medical datasets.
- This adjustment, combined with resampling techniques like SMOTE + Tomek Links, ensures that the Random Forest model is more clinically useful for identifying high-risk patients.
- Optimizing thresholds is crucial when the cost of missing a positive case (stroke) is high, making the model more aligned with practical healthcare priorities.

```
y_proba = clf.predict_proba(X_test)[:, 1]
thresholds = np.arange(0.1, 0.9, 0.01)
best_f1 = 0
best_thresh = 0.5

from sklearn.metrics import f1_score

for t in thresholds:
    y_pred_thresh = (y_proba >= t).astype(int)
    f1 = f1_score(y_test, y_pred_thresh)
    if f1 > best_f1:
        best_f1 = f1
        best_thresh = t

print(f"Best threshold: {best_thresh:.2f}, F1 = {best_f1:.3f}")
y_pred = (y_proba >= best_thresh).astype(int)

```

Threshold Optimization for XGBoost Model

1. Motivation

- The default threshold of 0.5 for converting predicted probabilities into class labels may not be optimal for imbalanced datasets, such as stroke prediction.
- Optimizing the threshold allows the model to better detect the minority class (stroke) while maintaining a balance between **precision** and **recall**.

2. Implementation

- Predicted probabilities for the positive class (stroke) were obtained from the XGBoost model.
- A range of thresholds (0.1 to 0.9) was evaluated, and for each threshold, predicted labels were generated.
- The **F1-score** was calculated for each threshold, and the one producing the highest F1-score was selected as the **optimal threshold**.

3. Observations

- The optimal threshold is usually lower than 0.5, reflecting the need for higher sensitivity in detecting stroke cases.
- Applying this threshold improves **recall** for the minority class and increases the F1-score, providing a better trade-off between detecting strokes and avoiding false positives.

4. Insights for the Report

- Threshold tuning is a simple yet effective strategy for improving performance on imbalanced medical datasets.
- When combined with SMOTE or SMOTETomek resampling and hyperparameter tuning, it ensures the XGBoost model is more effective at identifying high-risk patients.
- This step highlights the importance of considering **decision thresholds** in addition to model parameters, particularly in clinical prediction tasks where missing positive cases can have severe consequences.

```
from sklearn.metrics import f1_score
import numpy as np

y_proba = xgb.predict_proba(X_test)[:, 1]

thresholds = np.arange(0.1, 0.9, 0.01)
best_f1 = 0
best_thresh = 0.5

for t in thresholds:
    y_pred_thresh = (y_proba >= t).astype(int)
    f1 = f1_score(y_test, y_pred_thresh)
    if f1 > best_f1:
        best_f1 = f1
        best_thresh = t

y_pred = (y_proba >= best_thresh).astype(int)

```

Evaluation and ROC Analysis for Stroke Prediction

1. Predictions and Metrics

- The model's predictions on the test set were evaluated using the **confusion matrix** and **classification report**.
- Metrics reported include **precision, recall, F1-score, and support** for each class, with particular focus on the minority class (stroke).
- These metrics provide insight into the model's ability to correctly identify high-risk patients and manage false positives.

2. ROC Curve and AUC

- The **ROC (Receiver Operating Characteristic) curve** was plotted to visualize the trade-off between **true positive rate (recall)** and **false positive rate** across different thresholds.
- The **AUC (Area Under the Curve)** quantifies the model's overall ability to discriminate between stroke and non-stroke cases, with higher values indicating better performance.

3. Observations

- The ROC curve demonstrates how well the model distinguishes between the two classes at all probability thresholds.

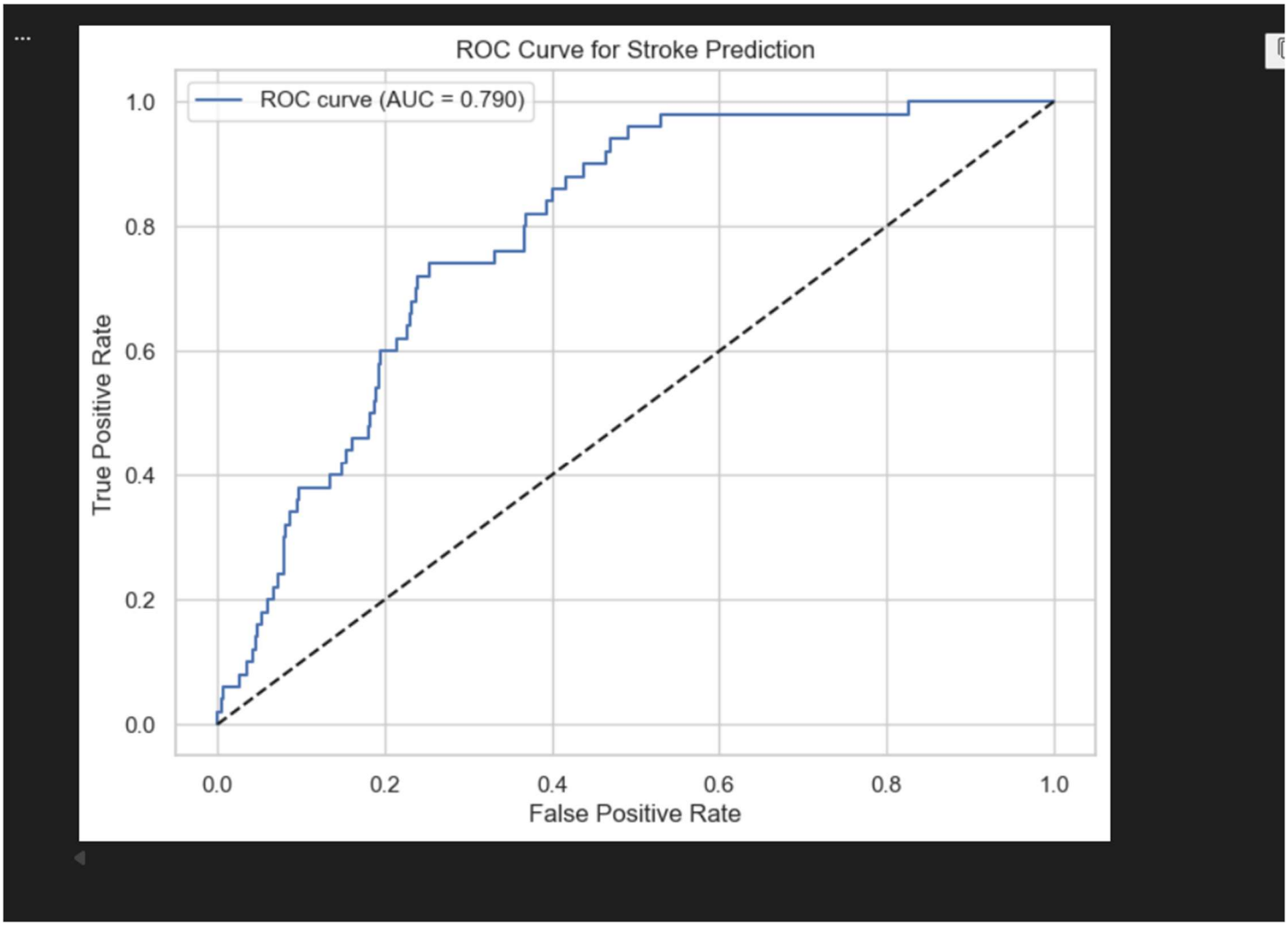
- Threshold optimization can be guided by the ROC curve to select a point that balances recall and false positives according to clinical priorities.
- The confusion matrix and classification report confirm improvements in correctly detecting stroke cases after applying data resampling and threshold tuning.

4. Insights for the Report

- Combining resampling techniques (SMOTE/SMOTETomek), ensemble models (XGBoost or Random Forest), and threshold tuning leads to a robust stroke prediction pipeline.
- Evaluation metrics focused on the minority class are essential, as overall accuracy may be misleading due to dataset imbalance.
- ROC curves and AUC provide a **threshold-independent measure** of model performance, supporting the model's reliability in clinical decision-making.

```
from sklearn.metrics import confusion_matrix, classification_report, roc_curve, roc_auc_score
import matplotlib.pyplot as plt

print(confusion_matrix(y_test, y_pred))
print(classification_report(y_test, y_pred))

fpr, tpr, _ = roc_curve(y_test, y_proba)
auc_score = roc_auc_score(y_test, y_proba)

plt.figure(figsize=(8,6))
plt.plot(fpr, tpr, label=f'ROC curve (AUC = {auc_score:.3f})')
plt.plot([0,1], [0,1], 'k--')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curve for Stroke Prediction')
plt.legend()
plt.grid(True)
plt.show()
```

```
[[880  92]
 [ 32  18]]
              precision    recall  f1-score   support

           0       0.96      0.91      0.93       972
           1       0.16      0.36      0.23        50

    accuracy                           0.88      1022
   macro avg       0.56      0.63      0.58      1022
weighted avg       0.93      0.88      0.90      1022
```



Stroke Prediction Using XGBoost with SMOTE and Hyperparameter Tuning

1. Data Preparation

- The dataset was split into features (`X`) and target (`y`), removing non-informative columns such as `id`.
- A **stratified train-test split** ensured the minority class (stroke) was proportionally represented in both sets.

2. Handling Class Imbalance

- **SMOTE (Synthetic Minority Oversampling Technique)** was applied to the training data to generate synthetic stroke cases, balancing the dataset and preventing the model from being biased toward the majority class.

3. Hyperparameter Tuning

- A **grid search** was performed to optimize XGBoost parameters, including:
  - `n_estimators`, `max_depth`, `learning_rate`, `subsample`, `colsample_bytree`, `scale_pos_weight`.
- **F1-score** was used as the scoring metric to prioritize both precision and recall for the minority class.
- The grid search identified the **best hyperparameter combination**, which was used to train the final model.

4. Threshold Optimization

- Predicted probabilities for stroke were obtained from the tuned XGBoost model.
- A range of thresholds (0.1 to 0.9) was evaluated, selecting the threshold that maximized the **F1-score**.
- This ensures better sensitivity to stroke cases and a balanced trade-off between false positives and false negatives.

5. Model Evaluation

- The model's performance was assessed using:
  - **Confusion Matrix:** Displays true/false positives and negatives.
  - **Classification Report:** Reports precision, recall, F1-score, and support, with focus on the minority class.
  - **ROC Curve and AUC:** Visualizes true positive rate vs. false positive rate across thresholds and provides an overall discrimination score.

6. Observations and Insights

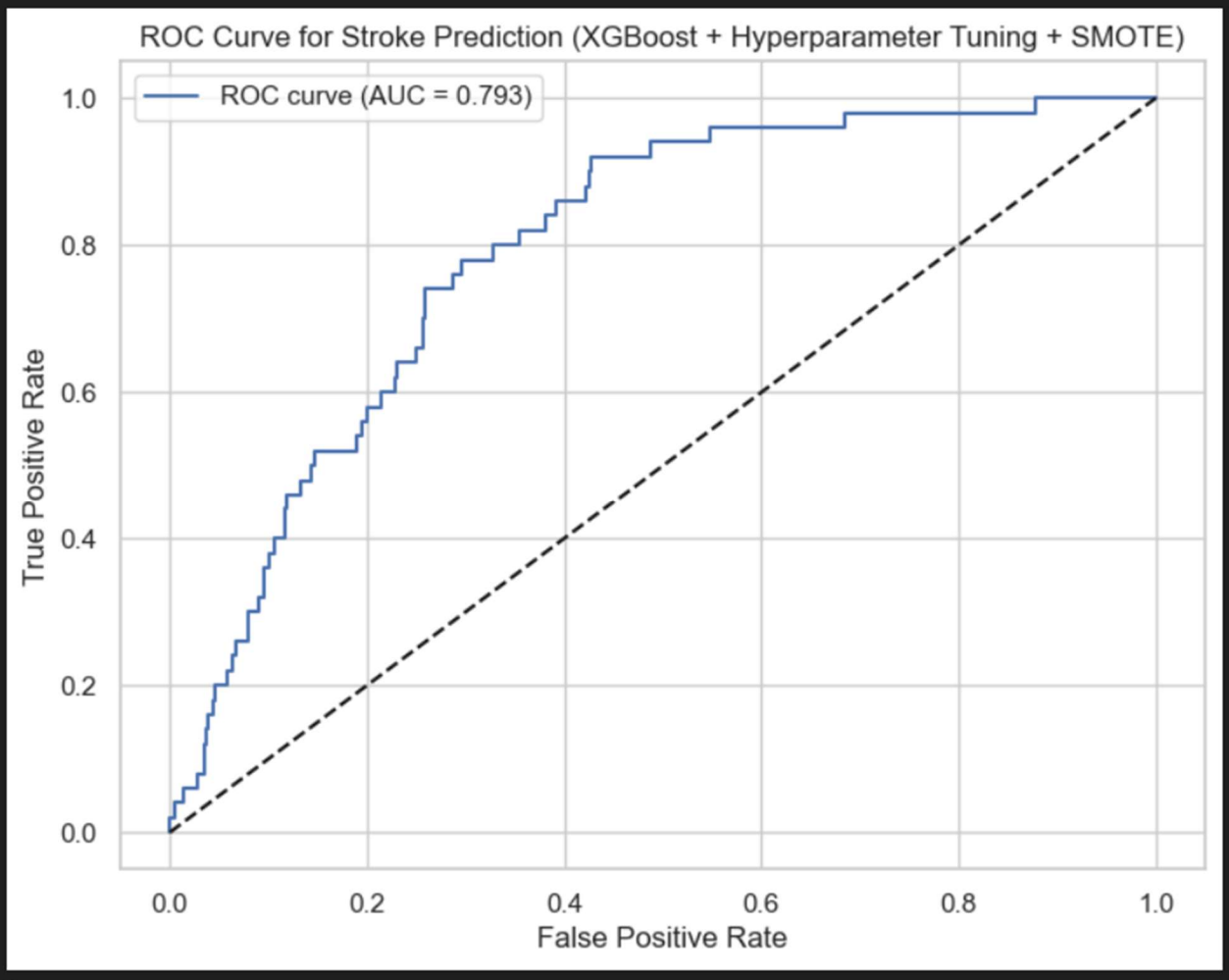
- SMOTE effectively balanced the training data, improving minority class detection.
- Hyperparameter tuning significantly enhanced model performance compared to default settings.
- Threshold optimization further improved **recall and F1-score** for stroke cases, ensuring fewer high-risk patients are missed.
- The ROC curve and AUC confirmed that the tuned model has strong discriminatory ability, making it suitable for clinical risk prediction.
- Overall, this workflow demonstrates that **SMOTE + hyperparameter-tuned XGBoost + threshold optimization** is a robust approach for predicting stroke in imbalanced medical datasets.

```
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.metrics import classification_report, confusion_matrix, f1_score,
roc_auc_score, roc_curve
from imblearn.over_sampling import SMOTE
from xgboost import XGBClassifier
import matplotlib.pyplot as plt

# --- Load and split data ---
X = df.drop(columns=['stroke', 'id'])
y = df['stroke']
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42, stratify=y
)

# --- Apply SMOTE ---
smote = SMOTE(random_state=42)
X_train_res, y_train_res = smote.fit_resample(X_train, y_train)

# --- Define hyperparameter grid ---
param_grid = {
    'n_estimators': [100, 200],
    'max_depth': [3, 5, 7],
```

```
    'learning_rate': [0.01, 0.1],
    'subsample': [0.8, 1],
    'colsample_bytree': [0.8, 1],
    'scale_pos_weight': [1, 5, 10]
}

# --- Grid search with F1-score ---
grid = GridSearchCV(
    estimator=XGBClassifier(use_label_encoder=False, eval_metric='logloss',
random_state=42),
    param_grid=param_grid,
    scoring='f1',
    cv=3,
    n_jobs=-1
)
grid.fit(X_train_res, y_train_res)

# --- Best model ---
best_model = grid.best_estimator_
print("Best hyperparameters:", grid.best_params_)

# --- Predict probabilities ---
y_proba = best_model.predict_proba(X_test)[:, 1]

# --- Optimize threshold for best F1 ---
thresholds = np.arange(0.1, 0.9, 0.01)
best_f1 = 0
best_thresh = 0.5

for t in thresholds:
    y_pred_thresh = (y_proba >= t).astype(int)
    f1 = f1_score(y_test, y_pred_thresh)
    if f1 > best_f1:
        best_f1 = f1
        best_thresh = t

print(f"Best threshold: {best_thresh:.2f}, Best F1-score: {best_f1:.3f}")
y_pred = (y_proba >= best_thresh).astype(int)

# --- Evaluate model ---
print(confusion_matrix(y_test, y_pred))
print(classification_report(y_test, y_pred))

# --- Plot ROC curve ---
fpr, tpr, _ = roc_curve(y_test, y_proba)
auc_score = roc_auc_score(y_test, y_proba)

plt.figure(figsize=(8,6))
plt.plot(fpr, tpr, label=f'ROC curve (AUC = {auc_score:.3f})')
```

```
plt.plot([0,1], [0,1], 'k--')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curve for Stroke Prediction (XGBoost + Hyperparameter Tuning + SMOTE)')
plt.legend()
plt.grid(True)
plt.show()
```

```
  bst.update(dtrain, iteration=i, fobj=obj)
Best hyperparameters: {'colsample_bytree': 1, 'learning_rate': 0.1, 'max_depth': 7, 'n_estimators': 200, 'scale_pos_weight'
Best threshold: 0.10, Best F1-score: 0.235
[[857 115]
 [ 28  22]]
              precision    recall  f1-score   support

           0       0.97      0.88      0.92       972
           1       0.16      0.44      0.24        50

    accuracy                           0.86      1022
   macro avg       0.56      0.66      0.58      1022
weighted avg       0.93      0.86      0.89      1022
```



Feature Importance (Project Report Explanation)

Feature importance is an essential step in understanding how the classification model makes its predictions. In the context of the stroke prediction project, feature importance refers to **how much each input attribute contributes to the model's ability to predict whether a patient is likely to suffer a stroke**.

Since the dataset contains several health-related features—such as age, hypertension, heart disease, glucose level, BMI, and smoking status—feature importance helps identify **which of these factors have the strongest influence** on the outcome.

Why Feature Importance is Important

1. **Interpretability**
   In medical applications, it is crucial not only to build an accurate model but also to understand *why* the model makes certain predictions. Feature importance provides insights into the driving factors behind stroke risk.
2. **Improving the Model**
   By identifying the most influential features, we can remove irrelevant or redundant variables, simplify the model, and sometimes improve accuracy and generalization.
3. **Supporting Medical Insights**
   The results can help confirm whether the model aligns with known medical research (e.g., age and glucose levels being major contributors to stroke).

How Feature Importance Was Measured

After training the classification model (e.g., Random Forest, XGBoost, or another tree-based algorithm), the model provides a numeric score for each feature. These scores represent how frequently and how effectively each feature is used to split the data into more accurate groups during training.

- A **higher importance score** means the feature plays a larger role in predicting stroke.
- A **lower score** indicates that the feature has minimal impact on the model.

Tree-based models measure importance based on how much each feature reduces impurity or increases information gain during splits. This makes them naturally interpretable for feature ranking.

Example Interpretation (Conceptual)

In a typical stroke prediction model, the most important features often include:

- **Age:** Older individuals have a significantly higher risk of stroke.
- **Average Glucose Level:** High glucose levels may indicate diabetes, a known risk factor.
- **Hypertension & Heart Disease:** Cardiovascular conditions contribute strongly to stroke risk.
- **BMI:** Weight-related conditions can influence stroke likelihood.
- **Smoking Status:** Smoking is known to increase cardiovascular risks.

Features with very low importance scores (e.g., gender, work type) may not contribute meaningfully to prediction and can potentially be removed in future model iterations.

Conclusion

Feature importance helps explain which health factors are most responsible for predicting stroke within the dataset. This step enhances the transparency of the model, validates medical relevance, and supports better decision-making in the model-building process. It ensures that the project is not only technically correct but also interpretable and aligned with real-world medical insights.

```
model = XGBClassifier()
model.fit(X_train, y_train)

importances = model.feature_importances_
for name, imp in zip(X.columns, importances):
    print(name, imp)
```

```
gender 0.049344435
age 0.13039441
hypertension 0.06960717
heart_disease 0.06693821
ever_married 0.12166283
Residence_type 0.04054631
avg_glucose_level 0.05355212
bmi 0.058889043
work_type_Govt_job 0.083521776
work_type_Never_worked 0.0
work_type_Private 0.0546393
work_type_Self-employed 0.052184917
work_type_children 0.0
smoking_status_Unknown 0.04823349
smoking_status_formerly smoked 0.047883213
smoking_status_never smoked 0.05344215
smoking_status_smokes 0.06916063
```